

Notas de aula

**MAC 0329**  
**Álgebra booleana e aplicações**

Nina S. T. Hirata

Departamento de Ciência da Computação  
Instituto de Matemática e Estatística  
Universidade de São Paulo

São Paulo, 2018

# Índice

<b>1</b>	<b>Introdução</b>	<b>6</b>
<b>2</b>	<b>Representação de números</b>	<b>10</b>
2.1	Sistemas de representação numérica . . . . .	10
2.2	Representação de números no computador . . . . .	10
<b>3</b>	<b>Funções e circuitos lógicos</b>	<b>11</b>
3.1	Adição e subtração . . . . .	11
3.2	Projeto de um somador . . . . .	15
3.3	Portas lógicas e circuitos lógicos . . . . .	19
<b>4</b>	<b>Álgebra booleana</b>	<b>25</b>
4.1	Definição axiomática de álgebra booleana . . . . .	26
4.2	Exemplos de álgebra booleana . . . . .	27
4.3	Princípio da dualidade . . . . .	31
4.4	Leis fundamentais da álgebra booleana . . . . .	32
<b>5</b>	<b>Expressões e Funções Booleanas</b>	<b>36</b>
5.1	Expressões Booleanas . . . . .	36
5.2	Funções booleanas . . . . .	37
5.3	Somas e produtos . . . . .	39

5.4	Formas canônicas . . . . .	41
5.4.1	Soma canônica de produtos (SOP canônica) . . . . .	42
5.4.2	Produto canônico de somas (POS canônica) . . . . .	45
<b>6</b>	<b>Minimização de funções booleanas</b>	<b>48</b>
6.1	Formas minimais . . . . .	48
6.1.1	Simplificação algébrica . . . . .	50
6.2	Mapas de Karnaugh . . . . .	51
6.2.1	Minimização usando mapas de Karnaugh . . . . .	52
6.2.2	Minimização na presença de don't cares . . . . .	55
6.3	Minimização de múltiplas funções . . . . .	56
6.4	PLA . . . . .	57
6.5	Método de Quine-McCluskey (*) . . . . .	60
6.5.1	Cálculo de implicants primos . . . . .	62
6.5.2	Funções incompletamente especificadas . . . . .	68
6.6	Outros algoritmos de minimização lógica 2-níveis . . . . .	68
<b>7</b>	<b>Circuitos combinacionais</b>	<b>70</b>
7.1	Somador . . . . .	70
7.2	Comparador . . . . .	70
7.3	Multiplexadores e demultiplexadores . . . . .	71
7.3.1	Multiplexadores . . . . .	71
7.3.2	Demultiplexadores . . . . .	73
7.3.3	Exemplos de utilização de MUX e DMUX . . . . .	73
7.4	Decodificadores e codificadores . . . . .	74
7.4.1	Decodificadores . . . . .	74
7.4.2	Realização multi-níveis de decodificadores* . . . . .	75

7.4.3 Codificadores . . . . .	77
7.4.4 Exemplos de utilização de codificadores e decodificadores . . . . .	78
7.5 Realização de funções arbitrárias . . . . .	80
7.5.1 Realização de funções com MUX . . . . .	80
7.5.2 Realização multi-níveis de funções com MUX . . . . .	80
7.5.3 Realização de funções com decodificadores . . . . .	83
<b>8 Lógica sequencial – memória</b>	<b>85</b>
8.1 Ondas digitais . . . . .	88
8.2 Latches <i>SR</i> . . . . .	90
8.3 <i>Flip-flop JK</i> . . . . .	94
8.4 <i>Flip-flops</i> mestre-escravo . . . . .	95
8.5 <i>Flip-flops</i> D e T . . . . .	97
<b>9 Circuitos sequenciais – Registradores e contadores</b>	<b>99</b>
9.1 Síncronos × Assíncronos . . . . .	99
9.2 Verificador de paridade em transmissão serial . . . . .	100
9.3 Registradores . . . . .	101
9.4 Contadores . . . . .	101
<b>10 Análise e projeto de circuitos sequenciais</b>	<b>107</b>
10.1 Análise de circuitos sequenciais . . . . .	108
10.2 Projeto de circuitos sequenciais . . . . .	113
<b>11 Organização de computadores</b>	<b>121</b>
11.1 O modelo de computação de von Neumann . . . . .	123
11.2 Ciclo de instrução ( <i>Fetch-Execute Cycle</i> ) . . . . .	125
11.3 Projeto e simulação de um processador . . . . .	125

11.4 Detalhamento do ciclo de instrução . . . . .	129
<b>A Relações de Ordem Parciais</b>	<b>131</b>
A.1 Conjuntos parcialmente ordenados (posets) . . . . .	131

# Capítulo 1

## Introdução

Ao interagirmos com um computador, utilizamos algum tipo de aplicativo. Por meio dos aplicativos é que conseguimos realizar atividades tais como ouvir música, assistir a um filme, fazer cálculos complicados, “chatear” com um amigo e assim por diante. Ao instalar novos aplicativos, a gama de atividades que podem ser realizadas pode ser ampliada.

De forma geral, ter noções sobre a organização e o funcionamento de um equipamento é importante para se fazer um bom uso do mesmo. Essa afirmação aplica-se por exemplo a carros, máquinas de lavar roupa, ou serras elétricas. Com computadores não é diferente.

Essas afirmações valem em relação a todos os usuários de computadores. Temos porém, dentre os usuários, os programadores que se dedicam a desenvolver os aplicativos. Há também aqueles que trabalham outros aspectos relevantes para o uso do computador (eficiência, usabilidade, segurança da informação, entre outros). Para esse grupo de usuários, ter conhecimentos (e não apenas noções) sobre a organização e funcionamento dos computadores é importante para que os aplicativos e outros componentes desenvolvidos sejam eficazes na realização das respectivas tarefas.

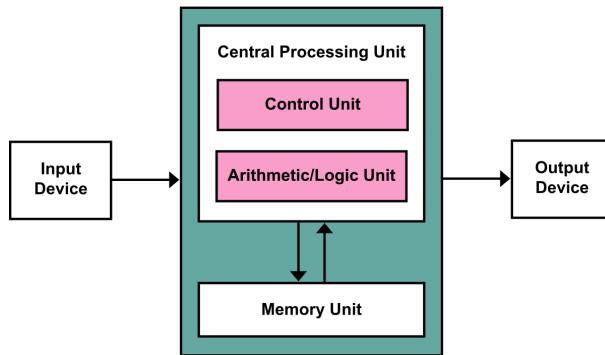
No estudo de computadores e computação, em geral fazemos a distinção entre *software* e *hardware*. Este último, *hardware*, refere-se à parte física do computador. Já o *software* refere-se à parte lógica que abarca, por exemplo, os programas de computador e o controle da execução desses programas.

Para entendermos como um computador é organizado e como ele funciona, é importante notarmos que eles são essencialmente máquinas para processamento de informação. Em termos de *hardware*, os componentes básicos de armazenamento são dispositivos físicos que conseguem representar e alterar entre dois estados, desligado e ligado. A esses dois estados associa-se os números 0 e 1, respectivamente. Portanto, as informações armazenadas e manipuladas por um

computador são representadas por uma sequência de 0s e 1s. Por conta disso, comumente ouvimos o termo “binário” quando o assunto é computação. O *hardware* do computador é projetado para armazenar dados e executar um conjunto de instruções; instruções essas que consistem de operações básicas como comparações e operações aritméticas, ou operação de cópia de dados de um local de armazenamento para outro, denominadas instruções de máquina.

Podemos dizer que um programa nada mais é do que uma sequência de instruções. O programador em geral escreve um programa utilizando-se de alguma linguagem de programação (tais como C, Java, Python, etc). Ou seja, as instruções de um programa não são instruções de máquina, são instruções expressas em uma linguagem mais próxima à linguagem natural. Logo, para que essas instruções possam ser executadas pelo *hardware*, elas precisam ser traduzidas para instruções de máquina. A este processo dá-se o nome de compilação ou interpretação. Na compilação, um outro programa é gerado, denominado programa executável ou programa binário, e então é esse programa que é de fato “executado” (i.e., colocado para que a sequência de instruções seja executada pelo *hardware*). Na interpretação, as instruções de máquina são geradas e enviadas para execução uma a uma. Não importa se compiladas ou interpretadas, os programas escritos em linguagem de programação são, portanto, convertidos para uma sequência de instruções de máquina que são as executadas de fato.

Como ocorre a execução de uma sequência de instruções ? Para isso, podemos nos apoiar em um modelo simples de computador: a arquitetura Von Neumann (ver figura 1.1). Além de



**Figura 1.1:** Arquitetura Von Neumann (fonte: *Wikipedia*).

dispositivos de entrada e de saída, o modelo de Von Newmann é composto por uma unidade de memória, e uma unidade de processamento central (CPU), que inclui uma unidade lógico-aritmética (ULA) e uma unidade de controle (UC). A memória é usada para o armazenamento de dados ou de instruções. A unidade lógico-aritmética é a parte do *hardware* que realiza as operações lógicas e aritméticas. Os dados operados pela ULA geralmente encontram-se na memória e os resultados da operação precisam ser armazenados também na memória. A UC é a parte do *hardware* que controla a execução das instruções, incluindo o fluxo de dados que ocorre, por exemplo entre a memória e a ULA. Para um programa ser executado no modelo Von Newmann, suas instruções de máquina devem ser armazenadas na memória e então executadas

uma a uma. A UC “sabe” qual é a próxima instrução a ser executada. Cada ciclo de execução de uma instrução consiste em buscar a instrução, decodificá-la e executá-la.

Um programa especial, chamado **sistema operacional** (SO), é importante nesse processo. Ao se ligar um computador, o SO é automaticamente colocado em execução. Quando um usuário inicia o uso de um aplicativo, cabe ao SO colocar as instruções de máquina correspondentes a esse aplicativo na memória do computador e indicar para a CPU onde se encontra a próxima instrução a ser executada. Além disso, o SO gerencia o uso de recursos do computador (tais como espaço em memória e tempo de CPU) de forma que vários aplicativos possam estar em execução “simultânea”. Cabe também a ele administrar interrupções (sinais que vêm de fora, tais como o do teclado, o do mouse, ou da rede, etc), ou capturar uma indicação de erro de execução e transmitir ao aplicativo.

Estas notas de aula reúnem conceitos e fundamentos importantes para o entendimento desse processo (execução de um programa em um computador), especificamente com respeito à parte relacionada à execução das instruções de máquina. Em algumas partes ela é bastante detalhada e completa, em outras é breve. Além disso, alguns tópicos não estão presentes (nesses casos, há indicação de referências bibliográficas a serem consultadas).

O conteúdo está organizado da seguinte forma. Note, no entanto, que o texto poderá ser atualizado durante o semestre e essa organização sofrer alterações.

- Representação de números em diferentes bases, especialmente a base 2 (binária). Representação de informação no computador, e em especial números inteiros sem e com sinal.
- Funções lógicas e circuitos digitais: operações aritméticas com números em binário e a realização do somador de bits por meio de um circuito lógico
- fundamentos de álgebra booleana, importantes para modelagem de funções lógicas, e portanto para projetar circuitos corretos e compactos
- Exemplos de circuitos combinacionais. Noções de modularização
- Componentes sequenciais: memória, registrador, contador. Clock. Máquina de estados.
- Organização do computador: como os componentes combinacionais (ULA), sequenciais (memória) e de controle controle interagem.

**Observação:** Uma sugestão de leitura (leve) para saber um pouco mais sobre vários dos termos mencionados acima (programa, instruções, binário, *hardware*, *software*, circuito lógico, etc) e como esses elementos se entrelaçam é o livro a seguir:



O livro apresenta também um panorama histórico da computação, desde a idade da pedra!

## Capítulo 2

# Representação de números

Objetivos desta parte do texto: entender a representação dos números em diferentes bases e como converter de uma base para outra; entender como os números inteiros são representados no computador, levando-se em conta que o número de *bits* (*BInary digiT*s) é fixo ( $n$ ); entender como representar números com sinal (isto é positivos e negativos).

**Observação:** este capítulo está praticamente vazio ... Como referência sugerimos:

- *A Tutorial on Data Representation Integers, Floating-point Numbers, and Characters*  
<http://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html>
- *CHAPTER 8 – Binary Addition and Two’s Complement* [http://chortle.ccsu.edu/assemblytutorial/Chapter-08/ass08\\_1.html](http://chortle.ccsu.edu/assemblytutorial/Chapter-08/ass08_1.html)

### 2.1 Sistemas de representação numérica

Sistemas de representação numérica: representações de números em diferentes bases (especialmente a base 2); conversão de representação de uma base para outra.

### 2.2 Representação de números no computador

Representação de números no computador: palavras “binárias” com número fixo de *bits* (tipicamente 64 ou 32); representação de números sem sinal e com sinal; representação complemento de dois; intervalo dos números que podem ser representados em  $n$  *bits*.

# Capítulo 3

## Funções e circuitos lógicos

Última atualização em 02/04/2017

Este capítulo visa introduzir noções iniciais sobre funções e circuitos lógicos. Mais precisamente, veremos que o processamento de informação pode ser modelado por funções lógicas e ser expresso por meio de circuitos. Para isso será utilizado um exemplo importante: as operações de adição e subtração, tanto no caso de inteiros sem sinal como no caso de inteiros com sinal (na notação complemento de dois).

### 3.1 Adição e subtração

Para realizar a adição de números binários, basta aplicarmos o mesmo algoritmo de adição que usamos no caso da adição de números na base 10. A única diferença é que no caso dos números binários há apenas dois dígitos possíveis. Assim, as adições possíveis de dois números de um dígito são:

$$\begin{array}{r} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ \hline - & - & - & - \\ 0 & 1 & 1 & 10 \end{array}$$

No último caso, há um vai-um para a segunda coluna. No caso da adição de números com múltiplos dígitos pode haver um vai-um vindo da coluna anterior. Assim, considerando-se os vai-uns, teríamos 8 possibilidades:

$$\begin{array}{ccccccccc} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \text{--> vai-uns} \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{--> termo 1} \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \text{--> termo 2} \end{array}$$

```
-- -- -- -- -- -- -- -- --> soma
0   1   1   10  1   10  10  11
```

Para a adição de números de múltiplos dígitos, basta que o procedimento de adição de *bits* seja repetido coluna a coluna, da direita para a esquerda, sem esquecer os vai-uns para a próxima coluna. Por exemplo, a adição de A=0101 e B=1001 seria:

```
1   ----> vai-uns
0101  ----> primeiro termo A
1001  ----> segundo termo B
-----
1110  ----> soma
```

Note que uma subtração pode ser calculada por meio de uma adição: calcular  $A - B$  é equivalente a calcular  $A + (-B)$ . Na representação complemento de dois o negativo de um número  $B$  corresponde a adicionar 1 ao complemento de  $B$ . Por exemplo, no caso de 4 bits temos que 7 é 0111 e -4 é 1100 (isto é, o complemento de 4, 1011, mais 0001). Logo, temos que  $7 - 4 = 7 + (-4) = 3$  em binário é:

```
11   ----> vai-uns
0111  ----> 7
1100  ----> -4
-----
0011  ----> 3 = 7 + (-4)
```

Ignorando-se o último carry, mais à esquerda, temos o resultado esperado, isto é, 3.

Um erro que pode ocorrer na adição de dois números é o *overflow*, situação na qual o resultado da adição não pode ser representado em  $n$  bits. São aqueles casos nos quais a soma é menor que  $-2^{n-1}$  ou maior que  $2^{n-1} - 1$ . Por exemplo, se  $n = 4$ , adições que resultam em -9 ou 8 são situações de *overflow*.

Um outro problema está em subtrações do tipo  $A - B$  nas quais o subtraendo  $B$  é exatamente igual a  $-2^{n-1}$ . Nesses casos, o valor  $-B$ , que é igual a  $2^{n-1}$ , não pode ser representado em  $n$  bits. Desta forma, a adição  $A + (-B)$  não será calculada corretamente.

Detectar *overflow* é importante para que o programa possa ser informado e realize o tratamento adequado no contexto da aplicação. Da mesma forma, tratar corretamente o problema  $B = -2^{n-1}$  na subtração  $A - B$  é importante para que a soma não difira do valor esperado.

Assim, para entender como essas questões são tratadas, consideraremos adições e subtrações no caso em que  $n = 2$ . Antes de mais nada, na tabela 3.1 são exibidos todos os binários de 2 bits, os respectivos complementos e o negativo, de acordo com a representação complemento de 2.

Note que o negativo de -2 (10 em binário) aparece como sendo ele próprio, o que está errado. Esse erro decorre do fato de não ser possível representar o número 2 (positivo) em dois bits. Por

base 10	base 2	complemento	negativo
0	00	11	00
1	01	10	11
-2	10	01	*10
-1	11	00	01

**Tabela 3.1:** Todos os binários de 2 bits, e respectivos complementos e negativos na notação complemento de dois. Note que  $-(-2) = 2$  não pode ser representado em 2 bits.

outro lado, uma vez que  $-B = \overline{B} + 1$ , em vez de calcular  $A - B = A + (-B)$ , podemos calcular  $A - B = A + (\overline{B} + 1)$ , evitando desta forma o problema de  $-B$  não poder ser representado corretamente.

### Adições envolvendo os valores 0, 1, -2, -1

$$0 + 0 = 0$$

$$\begin{array}{r} 00 \\ 00 \\ \hline 00 \end{array}$$

$$0 + 1 = 1$$

$$\begin{array}{r} 00 \\ 01 \\ \hline 01 \end{array}$$

$$0 + (-2) = -2$$

$$\begin{array}{r} 00 \\ 10 \\ \hline 10 \end{array}$$

$$0 + (-1) = -1$$

$$\begin{array}{r} 00 \\ 11 \\ \hline 11 \end{array}$$

$$1 + 1 = 2$$

$$\begin{array}{r} 1 \\ 01 \\ 01 \\ \hline 10 \end{array}$$

$$1 + (-2) = -1$$

$$\begin{array}{r} 01 \\ 10 \\ \hline 11 \end{array}$$

$$1 + (-1) = 0$$

$$\begin{array}{r} 11 \\ 01 \\ 11 \\ \hline 00 \end{array}$$

$$-2 + (-2) = -4$$

$$\begin{array}{r} 1 \\ 10 \\ 10 \\ \hline 00 \end{array}$$

$$-2 + (-1) = -3$$

$$\begin{array}{r} 1 \\ 10 \\ 11 \\ \hline 01 \end{array}$$

$$-1 + (-1) = -2$$

$$\begin{array}{r} 11 \\ 11 \\ 11 \\ \hline 10 \end{array}$$

Adições

**Subtrações envolvendo os valores 0, 1, -2, -1:** Como já mencionamos, a subtração  $A - B$  pode ser calculada fazendo-se  $A + (\overline{B} + 1)$ . Rearranjando esta última expressão como  $1 + A + \overline{B}$ , podemos considerar o 1 como sendo vai-um inicial, na coluna mais à direita, e realizar a soma  $A + \overline{B}$ . Com isso evita-se o problema da impossibilidade de se representar  $-(-2)$  em dois bits. Veja, por exemplo, como é feita a subtração  $-2 - (-2) = -2 + (2) = 0$ .

$$0 - 0 = \\ 0 + (-0) = 0$$

1	1	1
0	0	
1	1	
0	0	

$$0 - 1 = \\ 0 + (-1) = -1$$

0	1
0	0
1	0
1	1

$$0 - (-2) = \\ 0 + (2) = \textcolor{red}{2}$$

1	1
0	0
0	1
1	0

$$0 - (-1) = \\ 0 + (1) = 1$$

0	1
0	0
0	0
0	1

$$1 - 1 = \\ 1 + (-1) = 0$$

1	1	1
0	1	
1	0	
0	0	

$$1 - (-2) = \\ 1 + (2) = \textcolor{red}{3}$$

1	1
0	1
0	1
1	1

$$1 - (-1) = \\ 1 + (1) = \textcolor{red}{2}$$

1	1
0	1
0	0
1	0

$$-2 - (-2) = \\ -2 + (2) = 0$$

1	1	1
1	0	
0	1	
0	0	

$$-2 - (-1) = \\ -2 + (1) = -1$$

1
0
0
1

$$-1 - (-1) = \\ -1 + (1) = 0$$

1	1	1
1	1	
0	0	
0	0	

Subtrações

Quanto ao *overflow*, os casos em que ele ocorre são indicados em vermelho (tanto nas adições como nas subtrações). A deteção pode ser baseada no fato de a soma de dois positivos não poder ser negativa e, similarmente, de a soma de dois negativos não poder ser positiva (conforme destacados em azul e vermelho). Embora os exemplos anteriores sejam para números de 2 *bits*, não deverá ser difícil perceber que esse fato pode ser verificado para números com qualquer quantidade de *bits*. Além disso, podemos ver também que os casos de *overflow* são exatamente aqueles nos quais o *carry* na segunda coluna difere do *carry* na terceira coluna (confira esse fato em todos os casos).

**Resumo:** constatamos que a notação complemento de dois é versátil pois permite que subtrações sejam realizadas por meio de um algoritmo de adição. Os únicos ajustes necessários são:

- colocar o *carry* inicial (coluna 0) em 1 (nas adições ele é sempre 0)
- complementar o subtraendo (o termo a ser subtraído)

Verificamos também que a ocorrência de *overflow* nas operações de adição e subtração usando a notação complemento de dois pode ser detectada com uma verificação simples. Basta compararmos o *carry* na coluna  $n - 1$  com o *carry* na coluna  $n$ . Se eles diferirem, significa que houve *overflow*.

**Exercício:** Utilize o mesmo algoritmo acima para realizar a adição e subtração no caso de inteiros sem sinal. Neste caso, os números que podem ser representados em  $n = 2$  bits são 0, 1, 2 e 3. Assim, devemos ter:

0+0=0			
0+1=1	1+1=2		
0+2=2	1+2=3	2+2=overflow	
0+3=3	1+3=overflow	2+3=overflow	3+3=overflow
0-0=0	1-0=1	2-0=2	3-0=3
0-1=overflow	1-1=0	2-1=1	3-1=2
0-2=overflow	1-2=overflow	2-2=0	3-2=1
0-3=overflow	1-3=overflow	2-3=overflow	3-3=0

Como pode ser detectado o *overflow*?

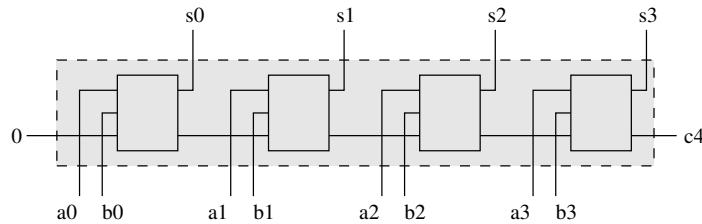
## 3.2 Projeto de um somador

Vimos que as subtrações podem ser realizadas por meio de adições. Além disso, como vimos anteriormente, o algoritmo de adição de números binários é similar à adição usual de números na base 10 (ou seja, os números são somados coluna a coluna, da direita para a esquerda) com a diferença de que há apenas dois possíveis dígitos (0 e 1).

Logo, se o computador possuir um componente capaz de realizar a adição dos bits referentes a uma coluna, então seria possível implementarmos um somador de inteiros representados em  $n$  bits. Bastaria ligarmos vários somadores de bits em cascata, para realizar as somas coluna a coluna, da coluna do bit menos significativo até a coluna do bit mais significativo.

**Somador (para números de 4 bits):** Sejam  $A = a_3 a_2 a_1 a_0$  e  $B = b_3 b_2 b_1 b_0$  dois números binários de 4 bits (onde o subscrito 0 e 3 representam, respectivamente, o bit menos e mais significativo). Suponha que temos componentes somadores de bits, que realizam a soma conforme descrito anteriormente. Denotando as entradas dos somadores de bits por  $a_i$ ,  $b_i$  e  $c_i$  (vai-um vindo da coluna anterior) e as saídas por  $s_i$  e  $c_{i+1}$ , a ligação em cascata pode ser feita de forma que a saída vai-um de uma coluna  $i$  alimente a entrada vai-um da coluna  $i + 1$ , como mostrado na figura 3.1.

Vamos então examinar como poderia ser construído um **somador de bits**. Seja  $c_{in}$  (*carry-in*, o vai-um que veio da coluna anterior) e  $a$  e  $b$  os bits dos números binários  $A$  e  $B$  sendo somados. A adição resulta em um bit soma  $s$  que corresponde à soma binária  $c_{in} + a + b$  e poderá gerar *carry-out*  $c_{out}$  (vai-um para a próxima coluna). Usando essas notações, podemos



**Figura 3.1:** Esquema de um somador de 4 bits. Cada retângulo corresponde a um componente somador de bits, cujos pinos à esquerda representam as entradas  $a_i$ ,  $b_i$  e  $c_i$  e os pinos à direita representam as saídas  $s_i$  e  $c_{i+1}$ . No diagrama  $c_0 = 0$ , adequado para a realização da operação de adição.

escrever o comportamento da soma referente a uma coluna, conforme a tabela 3.2. Este mesmo comportamento já foi discutido acima, no início da seção 3.1.

Entrada			Saída	
$c_{in}$	$a$	$b$	$s$	$c_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

**Tabela 3.2:** Tabela que define a soma dos dígitos binários em uma coluna, considerando-se a adição de dois números binários.

Essa tabela define, para todas as possíveis atribuições de valores às variáveis  $c_{in}$ ,  $a$  e  $b$ , o valor do bit soma  $s$  e do carry-out  $c_{out}$ . Em outras palavras, define uma função binária com três entradas e duas saídas, que descreve o comportamento de um **somador de bits**.

A operação de adição, assim como qualquer processamento que mapeia um conjunto de entradas para um conjunto de saídas, pode ser expressa por meio de tabelas como a acima. Por outro lado, uma tabela como a acima define uma função. Uma função que mapeia um certo número  $n$  de entradas binárias, que denotaremos  $x_1, x_2, \dots, x_n$ , para  $m$  saídas binárias  $y_1, \dots, y_m$ , pode ser denotada por  $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ . Podemos também interpretar  $f$  como  $f = (f_1, f_2, \dots, f_m)$ , com  $f_i : \{0, 1\}^n \rightarrow \{0, 1\}$  e  $y_i = f_i(x_1, x_2, \dots, x_n) \forall i$ .

**Breve pausa:**  $\{0, 1\}^n$  denota o produto cartesiano de conjuntos. Por exemplo,  $\{0, 1\}^3$  corresponde ao produto cartesiano  $\{0, 1\} \times \{0, 1\} \times \{0, 1\}$ . Um elemento de  $\{0, 1\} \times \{0, 1\} \times \{0, 1\}$  é uma trinca  $(a, b, c)$ , tal que  $a, b, c \in \{0, 1\}$ . De forma geral, dados por exemplo dois conjuntos  $A$

e  $B$ , temos que o produto cartesiano de  $A$  e  $B$  é o conjunto definido por  $A \times B = \{(a, b) : a \in A \text{ e } b \in B\}$  (isto é, pares  $(a, b)$  tais que o primeiro elemento do par pertence a  $A$  e o segundo elemento do par pertence a  $B$ ).

**Pergunta:** Se supormos que  $A$  contém  $n_a$  elementos e  $B$  contém  $n_b$  elementos, qual é o número de elementos (pares) no conjunto  $A \times B$ ?

---

**Exemplo:** Seja um processador de 8 bits. Em uma operação de adição, cada número a ser somado consiste de um conjunto de 8 bits. Considerando dois números a serem somados e mais o *carry* inicial, teríamos 17 entradas. O resultado da adição deverá ser armazenado também em 8 bits. Além disso, pode ser do interesse detectar *overflow*. Para isso deve-se ter acesso aos dois últimos *carries*. Portanto, a função de adição com essas características seria da forma  $f : \{0, 1\}^{17} \rightarrow \{0, 1\}^{10}$ .

Essa “transformação” de entrada para saída da operação de adição de dois números de 8 bits pode também ser descrita em uma tabela como a acima (não tentemos fazer isso, pois o número de linhas da tabela é muito grande. Quantas linhas terá a tabela?)

Considerando as saídas em função das entradas, no caso do somador de bits temos as funções  $s(c_{in}, a, b)$  e  $c_{out}(c_{in}, a, b)$ , ambas com 3 entradas. Usando a notação anteriormente mencionada, poderíamos também considerar que temos uma função  $f(c_{in}, a, b) = (s(c_{in}, a, b), c_{out}(c_{in}, a, b))$ .

Note que a saída  $s$  será igual a 1 se e somente se a soma  $c_{in} + a + b$  for ímpar. Isto pode ser escrito como:

$$s = 1 \iff (c_{in} + a + b) \% 2 = 1$$

na qual  $\%$  denota a operação “resto da divisão”.

Similarmente, a saída  $c_{out}$  será igual a 1 se e somente se a soma  $c_{in} + a + b$  for maior ou igual a 2. Isto pode ser escrito como:

$$s = 1 \iff (c_{in} + a + b) // 2 = 1$$

na qual  $//$  denota a operação “parte inteira da divisão”.

Ao considerarmos uma possível implementação da função acima no computador, essas expressões não são muito úteis pois o que estamos tentando implementar é justamente a operação de adição (e as expressões acima envolvem a operação de adição ...). Observe, porém, que podemos escrever a relação entrada-saída da função  $s$  de forma mais direta:

$$s(c_{in}, a, b) = 1 \iff (c_{in} = 0 \text{ e } a = 0 \text{ e } b = 1) \text{ ou } (c_{in} = 0 \text{ e } a = 1 \text{ e } b = 0) \text{ ou} \\ (c_{in} = 1 \text{ e } a = 0 \text{ e } b = 0) \text{ ou } (c_{in} = 1 \text{ e } a = 1 \text{ e } b = 1)$$

Note que no lado direito da equivalência, há uma expressão que inclui os **conectivos lógicos E e OU**, que podem ser “interpretados” da forma que estamos acostumados. A expressão “enumera” as entradas para as quais a função toma valor 1. Não é preciso muito esforço para verificar que para as demais entradas a função  $s$  toma valor 0. Essa expressão da função  $s$  em termos de conectivos lógicos E e OU sugere que se tivéssemos dispositivos físicos que implementassem o comportamento desses conectivos, seria possível implementar fisicamente a função  $s$ .

Vamos então definir esses conectivos:

		Conectivo lógico		
		E	OU	NÃO
$x_1$	$x_2$	$x_1 x_2$	$x_1 + x_2$	$\bar{x}_1$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

**Tabela 3.3:** Tabela-verdade das operações lógicas básicas. O valor 1 pode ser interpretado como verdadeiro (V) e o valor 0 como falso (F).

As colunas  $x_1, x_2$  denotam as entradas e as demais colunas definem funções lógicas correspondentes aos conectivos. Por exemplo, a terceira coluna define o E lógico que toma valor 1 se e somente se  $x_1 = 1$  e  $x_2 = 1$ . Isto pode ser expresso algebricamente por  $x_1 x_2$ . A expressão algébrica das demais funções aparece também no cabeçalho de cada coluna. Note que o símbolo  $+$ , usado para a operação OR (ou lógico), não é o mesmo da álgebra elementar; na álgebra elementar temos  $x + x = 2x$ , mas na álgebra booleana temos  $x + x = x$ . Esses detalhes serão vistos em um próximo capítulo.

**Exercícios:** Suponha que  $x_1 = 0$ ,  $x_2 = 1$  e  $x_3 = 1$

1. Qual é o valor da expressão  $x_1 + x_2 + x_3$  ?
2. Qual é o valor da expressão  $x_1 x_2 x_3$  ?
3. Qual é o valor da expressão  $x_1 x_2 + x_1 x_3$  ?
4. Qual é o valor da expressão  $x_1 x_2 + x_1 x_3 + x_2 x_3$  ?

5. Qual é o valor da expressão  $\overline{x_1}$  ?
6. Qual é o valor da expressão  $\overline{x_1 + x_2}$  ?

Retomando a expressão  $s$ , podemos dizer que ela toma valor 1 se ao menos uma das quatro conjunções, ( $c_{in} = 0$  e  $a = 0$  e  $b = 1$ ), ( $c_{in} = 0$  e  $a = 1$  e  $b = 0$ ), ( $c_{in} = 1$  e  $a = 0$  e  $b = 0$ ) ou ( $c_{in} = 1$  e  $a = 1$  e  $b = 1$ ), for verdade. Em outras palavras, a função que define  $s$  é tal que seu valor deve ser 1 para as entradas 011, 101, 110 e 111, e deve ser 0 para as demais entradas.

Observe que, por exemplo, quando  $c_{in} = 1$ ,  $a = 1$  e  $b = 1$ , a expressão  $c_{in} a b$  (produto das três variáveis, ou o E lógico das três variáveis) vale 1. Para qualquer outra atribuição de valor a essas três variáveis, o produto  $c_{in} a b$  toma valor 0. Podemos, portanto, fazer um raciocínio inverso e determinar qual é o produto que toma valor 1 para uma dada entrada específica. Por exemplo, qual é o produto que toma valor 1 quando  $c_{in} = 0$ ,  $a = 1$  e  $b = 1$ ? A variável  $c_{in}$  precisa aparecer barrada ( $\bar{c}_{in}$ ) no produto (pois caso contrário, a conjunção (E lógico) de  $c_{in}$  com qualquer outro valor seria necessariamente 0). O produto que queremos nesse caso é  $\bar{c}_{in} a b$ .

Ao fazermos o OU (disjunção) de todos os termos produtos associados a cada entrada que toma valor 1 na tabela, temos uma forma de expressar a função descrita pela tabela. No caso da função  $s$ , temos que:

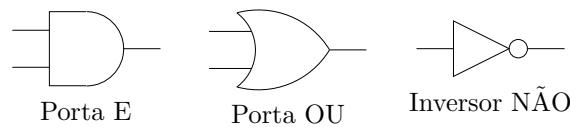
$$s(c_{in}, a, b) = \bar{c}_{in} \bar{a} b + \bar{c}_{in} a \bar{b} + c_{in} \bar{a} \bar{b} + c_{in} a b \quad (3.1)$$

Similarmente, temos que a função  $c_{out}$  pode ser escrita como:

$$c_{out}(c_{in}, a, b) = \bar{c}_{in} a b + c_{in} \bar{a} b + c_{in} a \bar{b} + c_{in} a b \quad (3.2)$$

### 3.3 Portas lógicas e circuitos lógicos

No caso de sistema digitais, os dispositivos que implementam os conectivos lógicos acima são denominados **portas lógicas**. A figura 3.2 mostra as **portas lógicas E** e **OU** e o **inversor NÃO**. Essas portas recebem sinais de entrada à esquerda e produzem um sinal de saída à direita.

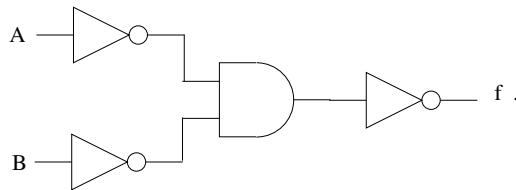


**Figura 3.2:** Representação gráfica de algumas portas lógicas.

Ao conectarmos a saída de um dispositivo nas entradas de outros, podemos construir uma rede interconexa de dispositivos. No caso de sistemas digitais, a interconexão define o que é chamado

de **círcuito digital**. Sem nos atermos à característica física da implementação, podemos usar genericamente o termo **círcuito lógico** para nos referirmos a essas redes interconexas.

Seja um circuito, como por exemplo o mostrado na figura 3.3, que usa três inversores e uma porta E. A saída do circuito, expressa como  $f(A, B)$ , indica que o valor da saída depende das duas entradas  $A$  e  $B$ , que podem ser vistas como variáveis da função.



**Figura 3.3:** Um circuito simples.

Dizemos que **um círcuito realiza uma função**. Podemos descrever seu funcionamento em uma tabela-verdade. Cada linha da tabela corresponde a uma das possíveis atribuições de valor às variáveis de entrada do circuito. No caso do circuito da figura 3.3, a tabela-verdade é mostrada na tabela 3.4, juntamente com os valores do circuito em pontos intermediários entre a entrada e a saída. Observe também que uma expressão algébrica que define a função pode ser obtida diretamente do circuito: primeiramente ambas as entradas são negadas e em seguida o resultado do E lógico entre elas é também negado, ou seja, temos  $f(A, B) = \overline{\overline{A} \overline{B}}$ .

$A$	$B$	$\overline{A}$	$\overline{B}$	$\overline{A} \overline{B}$	$f(A, B) = \overline{\overline{A} \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

**Tabela 3.4:** Tabela-verdade da função  $f(A, B) = \overline{\overline{A} \overline{B}}$ .

Note que  $f(A, B) = \overline{\overline{A} \overline{B}} = A + B$ . Isto mostra que uma mesma função pode ser realizada por diferentes circuitos. Quando dois circuitos realizam uma mesma função, eles são ditos equivalentes. O circuito da figura 3.3 é equivalente à porta OU.

Obter a tabela-verdade correspondente a um circuito é uma tarefa mecânica. Obviamente, as tabelas-verdade (e portanto as funções realizadas) por circuitos equivalentes são exatamente iguais. Por outro lado, o problema inverso de projetar um circuito que realiza uma dada função parece ser muito complexa. Porém, há uma forma sistemática que permite desenhar circuitos correspondentes a uma função binária qualquer. Essa forma sistemática baseia-se na mesma ideia da expressão escrita anteriormente para a saída  $s$  em termos dos conectivos E e OU, como veremos oportunamente.

Já vimos que uma função lógica pode ser descrita por meio de uma tabela-verdade (apesar de tal representação só ser praticável quando o número de entradas é pequeno ...). Similarmente,

a recíproca é verdadeira (isto é, uma tabela-verdade define uma função).

Reforçaremos aqui a ideia de que uma função (ou, equivalentemente, uma tabela-verdade) pode ser expressa por uma expressão, e que expressões por sua vez podem ser “traduzidas” para um circuito lógico. Buscaremos entender como esses elementos se relacionam.

Para tanto, voltemos para o exemplo do somador de *bits*, e em especial à função  $s$ . Por conveniência, abaixo reescrevemos a tabela-verdade do somador de bits e a expressão da saída  $s$  apresentada anteriormente:

Entrada			Saída	
$c_{in}$	$a$	$b$	$s$	$c_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Admitindo-se portas lógicas com múltiplas entradas, precisamos então de quatro portas E, uma porta OU e três inversores para implementar a função  $s$ , como mostrado à esquerda na figura 3.4.

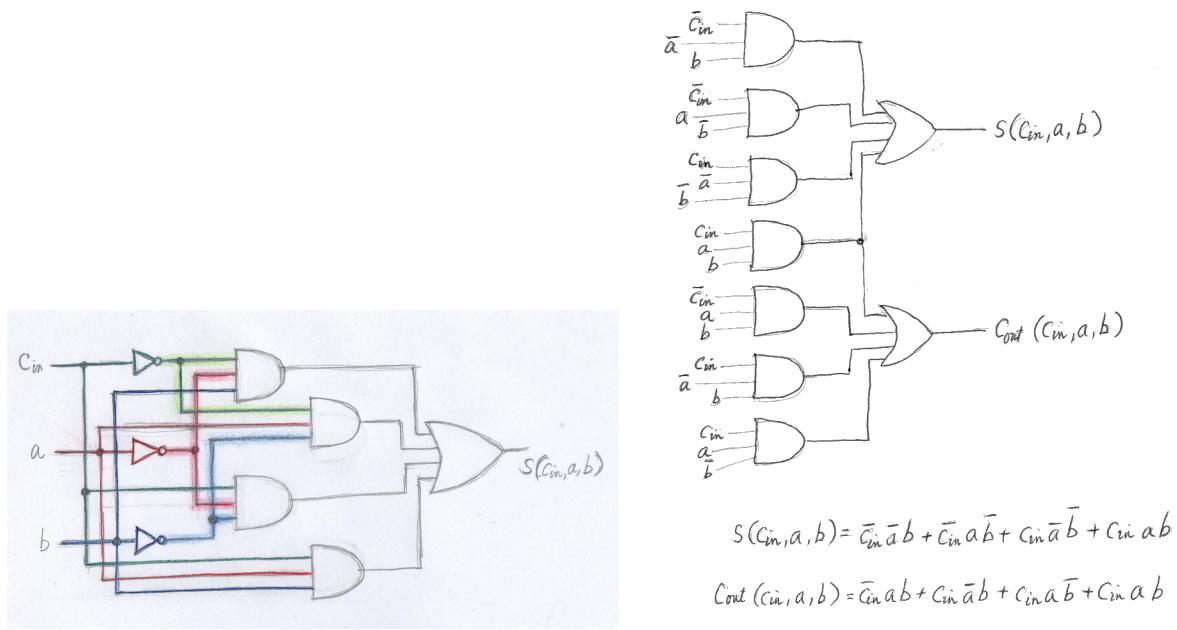
À direita da figura 3.4 é mostrado o circuito de ambas as funções,  $s$  e  $c_{out}$ . Como o último termo é o mesmo em  $s$  e em  $c_{out}$ , ele é compartilhado. São utilizados portanto 7 portas E (com três entradas cada) e 2 portas OU (com quatro entradas cada). Além disso, seriam necessários inversores, um para cada entrada. Daqui em diante, porém, em vez de desenhar inversores para as entradas barradas, iremos escrever diretamente a variável barrada na entrada do circuito.

Na figura 3.5 é mostrado um outro circuito que também realiza as duas funções  $s$  e  $c_{out}$ , porém com maior compartilhamento de subcircuitos.

Esse circuito utiliza a porta XOR, que corresponde ao OU EXCLUSIVO, definida e representada graficamente como segue:

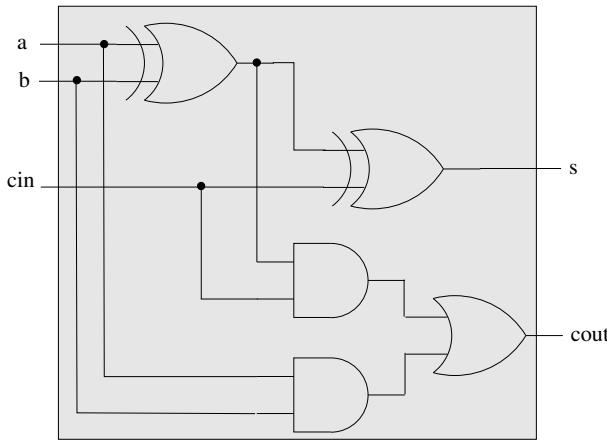
$x_1$	$x_2$	XOR ( $x_1 \oplus x_2$ )
0	0	0
0	1	1
1	0	1
1	1	0





**Figura 3.4:** À esquerda, circuito correspondente à expressão do bit soma  $s(c_{in}, a, b) = \bar{c}_{in} \bar{a} \bar{b} + \bar{c}_{in} a \bar{b} + c_{in} \bar{a} \bar{b} + c_{in} a b$ . À direita, Circuito somador de bits, implementando as funções  $s$  e  $c_{out}$  na forma soma de produtos.

**Exercício:** Escreva a tabela-verdade correspondente ao circuito da figura 3.5, incluindo uma coluna para a saída de cada uma das cinco portas lógicas (não apenas para as portas que geram as saídas do circuito). Verifique que as saídas  $s$  e  $c_{out}$  são conforme esperadas.



**Figura 3.5:** Porta XOR (ou exclusivo) à esquerda e esquema de um somador de *bits*, com compartilhamento de subcircuito, à direita.

A tabela está apresentada a seguir, na figura 3.6.

$c_{in}$	$a$	$b$	$y_1$	$y_2$	$y_3$	$s$	$c_{out}$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	1	0	0	1	0
0	1	1	0	0	1	0	1
1	0	0	0	0	0	1	0
1	0	1	1	1	0	0	1
1	1	0	1	1	0	0	1
1	1	1	0	0	1	1	1

$$s = y_1 \oplus c_{in} = (a \oplus b) \oplus c_{in}$$

$$c_{out} = y_2 + y_3 = y_1 c_{in} + ab = (a \oplus b) c_{in} + ab$$

**Figura 3.6:** Tabela-verdade do circuito da figura 3.5.

Como podemos ver, obtemos uma tabela igual ao da tabela 3.2. Isso significa que ambos os circuitos realizam a mesma função e portanto são equivalentes. As expressões de  $s$  e  $c_{out}$ , obtidas diretamente do circuito, são:

$$s = (a \oplus b) \oplus c \quad (3.3)$$

$$c_{out} = (a \oplus b) c_{in} + ab \quad (3.4)$$

Logo, é interessante investigar “Como podemos saber se duas expressões são equivalentes”. Será necessário escrever a tabela-verdade delas e compará-las ?

Uma das formas para se mostrar a equivalência de duas expressões é mostrando que uma pode ser derivada a partir da outra aplicando-se manipulações algébricas que não alteram o valor da expressão.

Adiantamos aqui um pouco do que veremos no próximo capítulo. Mostraremos como derivar as expressões nas equações 3.3 e 3.4 a partir das expressões nas equações 3.1 e 3.2. Para simplificar a notação, usamos  $c$  em lugar de  $c_{in}$  e organizamos as variáveis na ordem  $a, b, c$ .

$$\begin{aligned}
 s(a, b, c) &= \bar{a}b\bar{c} + a\bar{b}\bar{c} + \bar{a}\bar{b}c + abc \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\bar{a}\bar{b} + ab)c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\bar{a}\bar{b} + ab + 0 + 0)c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\bar{a}\bar{b} + ab + \bar{a}a + b\bar{b})c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\bar{a}a + \bar{a}\bar{b} + ba + b\bar{b})c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\bar{a}(a + \bar{b}) + b(a + \bar{b}))c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + ((\bar{a} + b)(a + \bar{b}))c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\overline{(\bar{a} + b)(a + \bar{b})})c \\
 &= (\bar{a}b + a\bar{b})\bar{c} + (\overline{a\bar{b} + \bar{a}b})c \\
 &= (a \oplus b)\bar{c} + (\overline{a \oplus b})c \\
 &= (a \oplus b) \oplus c
 \end{aligned}$$

Similarmente, para a equação do *carry-out* temos:

$$\begin{aligned}
 c_{out}(a, b, c) &= ab\bar{c} + \bar{a}bc + a\bar{b}c + abc \\
 &= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc \\
 &= (\bar{a}b + a\bar{b})c + ab(\bar{c} + c) \\
 &= (a \oplus b)c + ab
 \end{aligned}$$

## Capítulo 4

# Álgebra booleana

Última atualização em 03/04/2018

O termo “booleana” em álgebra booleana é em homenagem ao matemático inglês George Boole (1815-1864). George Boole foi um dos primeiros a considerar um tratamento sistemático do pensamento [Boole, 1854]. Em outras palavras, ele buscou criar um tratamento do raciocínio que fosse similar à representação de cálculos numéricos por meio de processamentos simbólicos. O sistema algébrico resultante é a **álgebra booleana**.

Além da aplicação em lógica, a álgebra booleana tem papel fundamental na modelagem de computadores eletrônicos. A relação entre álgebra booleana e sistemas digitais foi estabelecida por Claude Shannon na década de 1930, quando ele percebeu que as propriedades de circuitos de chaveamentos (redes interconexas de dispositivos do tipo liga-desliga) eram similares aos da álgebra booleana [Shannon, 1938].

Neste capítulo veremos uma definição formal de álgebra booleana, que é baseada em um conjunto de axiomas (ou postulados). Veremos também algumas leis ou propriedades de álgebras booleanas e que todas essas leis podem ser derivadas algebraicamente a partir dos postulados da definição. Alguns exemplos de álgebras booleanas são listados.

Referências para esta parte do curso: [Hill and Peterson, 1981], [Garnier and Taylor, 1992], [Whitesitt, 1961] entre outros.

## 4.1 Definição axiomática de álgebra booleana

Seja uma sétupla  $\langle A, +, \cdot, \bar{\cdot}, 0, 1 \rangle$  na qual  $A$  é um conjunto,  $+$  e  $\cdot$  são operações binárias sobre  $A$ ,  $\bar{\cdot}$  é uma operação unária em  $A$  e  $0$  e  $1$  são dois elementos distintos em  $A$ . O sistema algébrico  $\langle A, +, \cdot, \bar{\cdot}, 0, 1 \rangle$  é uma **álgebra booleana** se os seguintes axiomas são satisfeitos:

- A1. As operações  $+$  e  $\cdot$  são **comutativas**, ou seja, para todo  $x$  e  $y$  em  $A$ ,

$$x + y = y + x \quad \text{e} \quad x \cdot y = y \cdot x$$

- A2. Cada operação é **distributiva** sobre a outra, isto é, para todo  $x$ ,  $y$  e  $z$  em  $A$ ,

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \quad \text{e} \quad x + (y \cdot z) = (x + y) \cdot (x + z)$$

- A3. Os elementos  $0$  e  $1$  são os **elementos identidades**, ou seja, para todo  $x \in A$ ,

$$x + 0 = x \quad \text{e} \quad x \cdot 1 = x$$

- A4. Todo elemento  $x \in A$  possui um complemento, ou seja, existe um elemento  $\bar{x}$  em  $A$  tal que

$$x + \bar{x} = 1 \quad \text{e} \quad x \cdot \bar{x} = 0.$$

**Observação 1:** Na literatura encontramos outras definições para álgebra booleana. Em geral, as definições incorporam um maior número de propriedades. Vale registrar que os postulados acima apresentados, elaborados por Huntington em 1904, correspondem a um conjunto minimal de postulados, isto é, nenhum deles pode ser derivado a partir dos demais. Mais ainda, é um conjunto completo no sentido de que qualquer outra propriedade de uma álgebra booleana pode ser derivada a partir desses postulados. Desta forma, qualquer sistema algébrico que satisfaz os 4 axiomas acima é uma álgebra Booleana. Mais adiante mostraremos como a propriedade associativa (frequentemente incorporada à definição de álgebra booleana) e várias outras podem ser derivadas a partir dos postulados acima.

**Observação 2:** Pode-se fazer um paralelo com a álgebra elementar dos números. Por exemplo, sobre o conjunto dos números reais, define-se as operações de adição, subtração, etc. Essas operações satisfazem alguma propriedades (por exemplo, a adição é comutativa). Enquanto na álgebra booleana temos a noção de complemento, na álgebra elementar temos a noção de oposto (em relação à adição) e de inverso (em relação à multiplicação).

## 4.2 Exemplos de álgebra booleana

**Exemplo 1:** O conjunto  $B = \{0, 1\}$  com as definições

$$\bar{1} = 0 \quad \bar{0} = 1$$

$$1 \cdot 1 = 1 + 1 = 1 + 0 = 0 + 1 = 1$$

$$0 + 0 = 0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$$

é uma álgebra booleana.

Os axiomas A1, A3 e A4 são satisfeitos por definição. Para verificar o axioma A2, dados três elementos quaisquer  $x, y$  e  $z$  em  $B$ , podemos construir uma tabela verdade para todas as possíveis combinações de valores para  $x, y$  e  $z$ . Vejamos, nas colunas indicadas com \* na parte inferior da tabela, a validade da distributividade em relação a  $\cdot$ , ou seja, que  $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ .

$x$	$y$	$z$	$(y + z)$	$x \cdot (y + z)$	$(x \cdot y)$	$(x \cdot z)$	$(x \cdot y) + (x \cdot z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1
			*				*

Denotamos esta álgebra booleana por  $\langle B, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$ . Esta é a álgebra que está por trás dos circuitos lógicos.

---

**Exemplo 2:** O cálculo proposicional é um campo da lógica matemática que estuda proposições, ou seja, afirmações que ou são verdadeiras (V) ou são falsas (F), mas não ambas. As proposições podem ser conectadas usando-se os conectivos lógicos E, OU e NÃO, dando origem a novas proposições. Os conectivos lógicos podem ser representados pelos símbolos conforme tabela a seguir.

Conectivo	símbolo
E	$\wedge$
OU	$\vee$
NÃO	$\neg$

Supondo que  $x$  e  $y$  são duas proposições quaisquer, define-se essas operações conforme a tabela-verdade a seguir:

		$x$	$y$	$x \wedge y$			$x$	$y$	$x \vee y$
$x$	$\neg x$	F	F	F	F	F	F	F	F
F	V	F	V	F	F	V	F	V	V
V	F	V	F	F	V	F	V	V	V
		V	V	V	V	V	V	V	V

Qualquer semelhança com as operações lógicas vistas no contexto de circuitos lógicos não é mera coincidência. De fato, a lógica (ou cálculo) proposicional é uma álgebra booleana e ela tem uma correspondência um-para-um com  $\langle B, +, \cdot, \bar{\cdot}, 0, 1 \rangle$ , visto acima, conforme apontado a seguir.

Lógica proposicional	álgebra booleana $B$
$\vee$	$+$
$\wedge$	$\cdot$
F	0
V	1
$\neg x$	$\bar{x}$

Como consequência, temos também a correspondência entre as tabelas-verdade das operações  $\neg$ ,  $\vee$ ,  $\wedge$  com as tabelas-verdade das operações :  $\bar{\cdot}$ ,  $+$  e  $\cdot$ .

$x$	$y$	$\neg x$	$x \vee y$	$x \wedge y$	$x$	$y$	$\bar{x}$	$x + y$	$x \cdot y$
F	F	V	F	F	0	0	1	0	0
F	V	V	V	F	0	1	1	1	0
V	F	F	V	F	1	0	0	1	0
V	V	F	V	V	1	1	0	1	1

**Exemplo 3:** O conjunto  $B^n = B \times B \times \dots \times B$ , com as operações  $+$ ,  $\cdot$  e  $\bar{\cdot}$  herdadas de  $B$  e definidas, para quaisquer  $(x_1, x_2, \dots, x_n), (y_1, y_2, \dots, y_n) \in B^n$ , da seguinte forma

$$(x_1, x_2, \dots, x_n) + (y_1, y_2, \dots, y_n) = (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n)$$

$$(x_1, x_2, \dots, x_n) \cdot (y_1, y_2, \dots, y_n) = (x_1 \cdot y_1, x_2 \cdot y_2, \dots, x_n \cdot y_n)$$

$$\overline{(x_1, x_2, \dots, x_n)} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$$

e com

$$\mathbf{0} = (0, 0, \dots, 0)$$

$$\mathbf{1} = (1, 1, \dots, 1)$$

é uma álgebra booleana.

Mostremos que a segunda igualdade  $x + (y \cdot z) = (x + y) \cdot (x + z)$  do Axioma 2, distributividade, é satisfeita. Para simplificar a notação, omitiremos a operação  $\cdot$  e consideraremos  $n = 2$ , porém note que o mecanismo aplica-se a qualquer  $n > 1$ . Nas passagens a seguir, todas as igualdades, exceto a marcada com “A2” (axioma 2), são derivadas aplicando-se a definição das operações  $+$  ou  $\cdot$  definidas sobre  $B^2$ .

$$\begin{aligned} (x_1, x_2) + (y_1, y_2)(z_1, z_2) &= (x_1, x_2) + (y_1 z_1, y_2 z_2) \\ &= (x_1 + y_1 z_1, x_2 + y_2 z_2) \\ &\stackrel{A2}{=} ((x_1 + y_1)(x_1 + z_1), (x_2 + y_2)(x_2 + z_2)) \\ &= ((x_1 + y_1), (x_2 + y_2))((x_1 + z_1), (x_2 + z_2)) \\ &= ((x_1, x_2) + (y_1, y_2))((x_1, x_2) + (z_1, z_2)) \end{aligned}$$

Verifique que os demais axiomas também são satisfeitos.

---

**Exemplo 4:** Dado um conjunto  $S$ ,  $\mathcal{P}(S)$  denota o conjunto das partes de  $S$  (também chamado conjunto potência de  $S$ ), isto é,  $\mathcal{P}(S) = \{X : X \subseteq S\}$ . Se  $S$  possui  $k$  elementos, o conjunto  $\mathcal{P}(S)$  possui  $2^k$  elementos (por quê?).

Sobre conjuntos temos as operações bem conhecidas de união, interseção e complemento, denotados respectivamente por  $\cup$ ,  $\cap$  e  $(\cdot)^c$ . Temos que  $\langle \mathcal{P}(S), \cup, \cap, \cdot^c, \emptyset, S \rangle$  é uma álgebra booleana.

Conforme já estamos familiarizados, as propriedades da álgebra de conjuntos equivalentes aos 4 postulados da definição de álgebra booleana são:

$$A1. X \cup Y = Y \cup X \quad \text{e} \quad X \cap Y = Y \cap X$$

$$A2. X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z) \quad \text{e} \quad X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$$

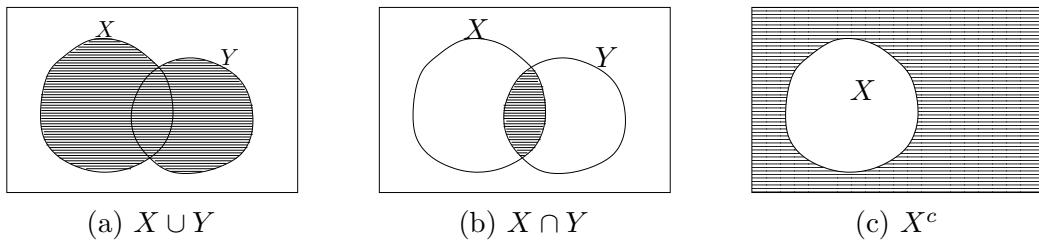
$$A3. \emptyset \cup X = X \quad \text{e} \quad U \cap X = X$$

$$A4. X \cap X^c = \emptyset \quad \text{e} \quad X \cup X^c = U$$

Dessas propriedades, a única que pode não ser trivial é a A2. Para se convencer da validade

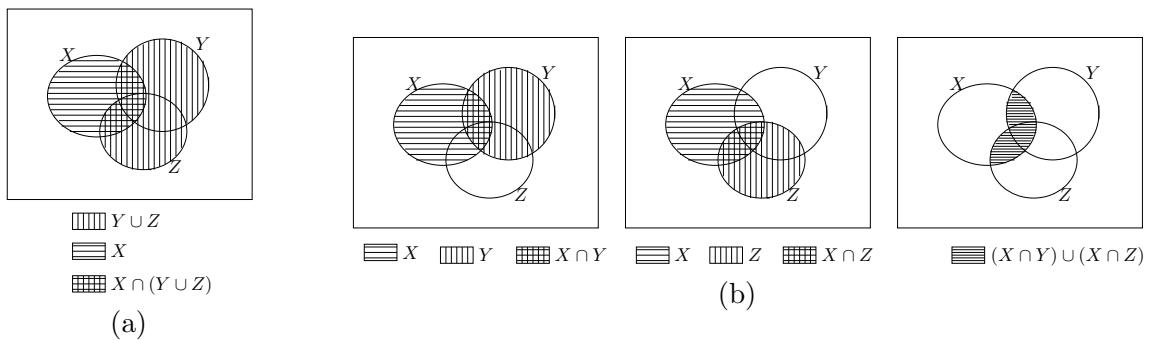
dessas proriedades, pode-se recorrer aos **diagramas de Venn**.

No diagrama de Venn, o conjunto universo é representado por um retângulo, mais precisamente, pelos pontos interiores ao retângulo. Qualquer conjunto é desenhado como sendo uma curva fechada, inteiramente contida no retângulo. Pontos interiores à curva correspondem aos elementos do conjunto. No exemplo da figura 4.1, a união e interseção de dois conjuntos genéricos estão representadas pelas regiões hachuradas das figuras 4.1a e 4.1b, respectivamente. O complemento de um conjunto é representado no diagrama da figura 4.1c.



**Figura 4.1:** Diagramas de Venn: (a) união de dois conjuntos, (b) interseção de dois conjuntos, e (c) complemento de um conjunto.

Como exemplo, vamos verificar a propriedade  $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$ . O conjunto  $X \cap (Y \cup Z)$  corresponde à região hachurada pelas linhas verticais e pelas linhas horizontais na figura 4.2a. Esta coincide com a região hachurada no diagrama mais à direita da figura 4.2b, que representa o conjunto  $(X \cap Y) \cup (X \cap Z)$ .



**Figura 4.2:** (a)  $X \cap (Y \cup Z)$  e (b)  $(X \cap Y) \cup (X \cap Z)$ .

**Observação:** Seja  $S$  um conjunto com  $n$  elementos. Note que qualquer subconjunto de  $S$  pode ser representado por um elemento de  $B^n$ . Por exemplo, se  $S = \{a, b, c\}$  então o subconjunto  $\{a, c\}$  pode ser representado por 101 e o subconjunto  $\{a\}$  por 100 e assim por diante.

### 4.3 Princípio da dualidade

Vimos que podemos escrever expressões envolvendo elementos de uma álgebra booleana  $A$ . Por exemplo,  $a b c + a b \bar{c}$  é uma forma para se expressar um elemento de  $A$  (da mesma forma que  $(2 + 3)*4$  expressa o 20 no caso de inteiros). As expressões podem ser manipuladas algebraicamente aplicando-se as propriedades da álgebra booleana (por exemplo,  $a b c + a b \bar{c} = a b(c + \bar{c}) = a b(1) = a b$ ). A expressão resultante após a aplicação sucessiva de propriedades representa o mesmo elemento representado pela expressão inicial.

Como as propriedades são verdadeiras com respeito a quaisquer elementos em  $A$ , quando escrevemos  $a b c + a b \bar{c} = a b(c + \bar{c}) = a b(1) = a b$ , significa que a identidade algébrica  $a b c + a b \bar{c} = a b$  vale para quaisquer três elementos  $a, b, c \in A$ .

O **dual** de uma expressão é obtida trocando-se todas as ocorrências de  $+$  por  $\cdot$ , todas as ocorrências de  $\cdot$  por  $+$ , todas as ocorrências de 0 por 1, e todas as ocorrências de 1 por 0.

O **princípio da dualidade** da álgebra booleana afirma que se uma identidade algébrica é válida então a identidade dual também é válida. Isto é, se conseguimos provar algebraicamente que  $E_1 = E_2$ , automaticamente teremos a prova de que  $dual(E_1) = dual(E_2)$ .

De fato, isso faz sentido pois o dual de cada uma das propriedades da definição de álgebra booleana é também uma propriedade presente na definição. Observe:

Axioma A1	$\begin{array}{rcl} x \cdot y & = & y \cdot x \\ \downarrow & & \downarrow \\ x + y & = & y + x \end{array}$
Axioma A2	$\begin{array}{rcl} x \cdot (y + z) & = & (x \cdot y) + (x \cdot z) \\ \downarrow & & \downarrow \\ x + (y \cdot z) & = & (x + y) \cdot (x + z) \end{array}$
Axioma A3	$\begin{array}{rcl} x + 0 & = & x \\ \downarrow & & \downarrow \\ x \cdot 1 & = & x \end{array}$
Axioma A4	$\begin{array}{rcl} x + \bar{x} & = & 1 \\ \downarrow & & \downarrow \\ x \cdot \bar{x} & = & 0 \end{array}$

**Obs.:** Uma confusão comum é pensar que uma expressão é equivalente ao seu dual. Isto em

geral não é verdade. Por exemplo, o dual de  $1 \cdot 1$  é  $0 + 0$ . Sabemos (pelo axioma A3) que  $1 \cdot 1 = 1 \neq 0 = 0 + 0$ .

## 4.4 Leis fundamentais da álgebra booleana

Desta parte em diante omitiremos o símbolo  $\cdot$  na maioria das vezes; em vez de  $x \cdot y$ , escreveremos simplesmente  $xy$ . Suponha que  $\langle A, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$  é uma álgebra booleana. Então, as seguintes propriedades são válidas.

**Unicidade do 0 e 1:** Os elementos 0 e 1 são únicos.

Sejam dois elementos zero,  $0_1$  e  $0_2$ . Por A3, temos que para todo  $x_1$  e  $x_2$  em  $A$ ,

$$x_1 + 0_1 = x_1 \quad \text{e} \quad x_2 + 0_2 = x_2$$

Logo, em particular para  $x_1 = 0_2$  e  $x_2 = 0_1$ , temos que

$$0_2 + 0_1 = 0_2 \quad \text{e} \quad 0_1 + 0_2 = 0_1$$

Por A1 e pela transitividade de  $=$ , resulta que  $0_1 = 0_2$ .

A unicidade de 1 pode ser provada usando o princípio da dualidade.

**Idempotência:** Para todo elemento  $x \in A$ ,  $x + x = x$  e  $xx = x$ .

$$\begin{array}{lll} x + x &= (x + x) \cdot 1 & (A3) \\ &= (x + x)(x + \bar{x}) & (A4) \\ &= x + x\bar{x} & (A2) \\ &= x + 0 & (A4) \\ &= x & (A3) \end{array} \quad \begin{array}{lll} xx &= xx + 0 & (A3) \\ &= xx + x\bar{x} & (A4) \\ &= x(x + \bar{x}) & (A2) \\ &= x \cdot 1 & (A4) \\ &= x & (A3) \end{array}$$

**Identidade:** Para todo  $x \in A$ ,  $x + 1 = 1$  e  $x0 = 0$ .

$$\begin{array}{lll} x + 1 &= 1 \cdot (x + 1) & (A3) \\ &= (x + \bar{x})(x + 1) & (A4) \\ &= x + \bar{x} \cdot 1 & (A2) \\ &= x + \bar{x} & (A3) \\ &= 1 & (A4) \end{array}$$

**Complemento do um (zero):**  $\bar{1} = 0$  e  $\bar{0} = 1$ .

$$\begin{aligned}\bar{1} &= \bar{1} \cdot 1 && (A3) \\ &= 0 && (A4)\end{aligned}$$

**Absorção:** Para todo  $x, y \in A$ ,  $x + xy = x$  e  $x(x + y) = x$ .

$$\begin{aligned}x + xy &= x \cdot 1 + xy && (A3) \\ &= x(1 + y) && (A2) \\ &= x \cdot 1 && (\text{Identidade}) \\ &= x && (A3)\end{aligned}$$

**Unicidade de  $\bar{x}$ :** O inverso de qualquer elemento  $x \in A$  é único.

Sejam  $\bar{x}_1$  e  $\bar{x}_2$  em  $A$  tais que

$$x + \bar{x}_1 = 1 \quad \text{e} \quad x + \bar{x}_2 = 1 \quad \text{e} \quad x\bar{x}_1 = 0 \quad \text{e} \quad x\bar{x}_2 = 0$$

Então, temos que

$$\begin{aligned}\bar{x}_2 &= 1 \cdot \bar{x}_2 && (A3) \\ &= (x + \bar{x}_1)\bar{x}_2 && (\text{hipótese}) \\ &= x\bar{x}_2 + \bar{x}_1\bar{x}_2 && (A2) \\ &= 0 + \bar{x}_1\bar{x}_2 && (\text{hipótese}) \\ &= x\bar{x}_1 + \bar{x}_1\bar{x}_2 && (\text{hipótese}) \\ &= (x + \bar{x}_2)\bar{x}_1 && (A2) \\ &= 1 \cdot \bar{x}_1 && (\text{hipótese}) \\ &= \bar{x}_1 && (A3)\end{aligned}$$

**Involução:** Para todo  $x \in A$ ,  $\bar{\bar{x}} = x$ .

Seja  $\bar{\bar{x}} = y$ . Então, por A4 temos que  $\bar{x}y = 0$  e  $\bar{x} + y = 1$ . Mas por A4,  $\bar{x}x = 0$  e  $\bar{x} + x = 1$ . Por causa da unicidade do complemento,  $\bar{\bar{x}} = y = x$ .

**Associatividade:** Para quaisquer  $x, y, z \in A$ ,  $x + (y + z) = (x + y) + z$  e  $x(yz) = (xy)z$ .

[Lema] Para quaisquer  $x, y, z \in A$ ,  $x[(x + y) + z] = [(x + y) + z]x = x$ .

$$\begin{aligned}
 x[(x+y)+z] &= [(x+y)+z]x \quad (A1) \\
 x[(x+y)+z] &= x(x+y)+xz \quad (A2) \\
 &= x+xz \quad (\text{absorção}) \\
 &= x \quad (\text{absorção})
 \end{aligned}$$

Usando o lema acima, provaremos a propriedade associativa. Seja

$$\begin{aligned}
 Z &= [(x+y)+z][x+(y+z)] \\
 &= [(x+y)+z]x + [(x+y)+z](y+z) \quad (A2) \\
 &= x + [(x+y)+z](y+z) \quad (\text{lema}) \\
 &= x + \{(x+y)+z\}y + \{(x+y)+z\}z \quad (A2) \\
 &= x + \{(y+x)+z\}y + \{(x+y)+z\}z \quad (A1) \\
 &= x + \{y + [(x+y)+z]z\} \quad (\text{lema}) \\
 &= x + (y+z)
 \end{aligned}$$

De forma similar,

$$\begin{aligned}
 Z &= (x+y)[x+(y+z)] + z[x+(y+z)] \quad (A2) \\
 &= (x+y)[x+(y+z)] + z \quad (\text{lema}) \\
 &= \{x[x+(y+z)] + y[x+(y+z)]\} + z \quad (A2) \\
 &= \{x[x+(y+z)] + y\} + z \quad (\text{lema}) \\
 &= (x+y) + z \quad (\text{lema})
 \end{aligned}$$

Logo,  $x + (y+z) = (x+y) + z$

**Teorema de DeMorgan** Para quaisquer  $x, y \in A$ ,  $\overline{(x+y)} = \overline{x}\overline{y}$  e  $\overline{xy} = \overline{x} + \overline{y}$ .

Vamos mostrar que  $(x+y) + \overline{x}\overline{y} = 1$  e que  $(x+y)(\overline{x}\overline{y}) = 0$ .

$$\begin{aligned}
 (x+y) + \overline{x}\overline{y} &= [(x+y) + \overline{x}][(x+y) + \overline{y}] \quad (A2) \\
 &= [\overline{x} + (x+y)][\overline{y} + (x+y)] \quad (A1) \\
 &= [(\overline{x}+x)+y][x+(\overline{y}+y)] \quad (\text{Associativa} + A1) \\
 &= 1 \cdot 1 \quad (A4 + \text{Identidade}) \\
 &= 1 \quad (A3)
 \end{aligned}$$

$$\begin{aligned}
 (x+y) \cdot \overline{x}\overline{y} &= x(\overline{x}\overline{y}) + y(\overline{y}\overline{x}) \quad (A2 + A1) \\
 &= (x\overline{x})\overline{y} + (y\overline{y})\overline{x} \quad (\text{associativa}) \\
 &= 0 + 0 \quad (A4 + \text{Identidade}) \\
 &= 0 \quad (A3)
 \end{aligned}$$

Portanto, pela unicidade do complemento, podemos concluir que  $\overline{(x+y)} = \overline{x}\overline{y}$ .

A igualdade dual pode ser demonstrada pelo princípio da dualidade, ou usando o fato de que as igualdades acima valem também para  $\overline{x}$  e  $\overline{y}$  no lugar de  $x$  e  $y$ .  $\square$

**Exercícios:**

- Mostre que o conjunto  $B^n$  mais as operações definidas no exemplo 4 da página 28 é uma álgebra booleana.
- Considere o conjunto dos números reais  $\mathbb{R}$ , juntamente com as operações usuais de adição e multiplicação. Quais dos axiomas A1, A2, A3 não são satisfeitos? É possível definir uma operação unária em  $\mathbb{R}$  tal que o axioma A4 seja satisfeito?
- Seja  $A = \{1, 2, 3, 5, 6, 10, 15, 30\}$ , ou seja, o conjunto de divisores de 30. Defina operações binárias  $+$  e  $\cdot$  e uma operação unária  $\bar{\phantom{x}}$  da seguinte forma: para qualquer  $a_1, a_2 \in A$ ,

$$a_1 + a_2 = \text{o mínimo múltiplo comum entre } a_1 \text{ e } a_2$$

$$a_1 \cdot a_2 = \text{o máximo divisor comum entre } a_1 \text{ e } a_2$$

$$\bar{a}_1 = 30/a_1$$

Quais são os elementos identidade com respeito a  $+$  e  $\cdot$ ? Mostre que  $A$ , com as três operações acima, é uma álgebra booleana.

- Prove, algebricamente, as seguintes igualdades
  - $x + \bar{x}y = x + y$  (e sua dual  $x(\bar{x} + y) = xy$ )
  - $x + y = \bar{\bar{x}}\bar{y}$  (e sua dual  $xy = \bar{x}\bar{y}$ )
  - $(x + y)(x + \bar{y}) = x$  (e sua dual  $xy + x\bar{y} = x$ )
  - (Teorema do consenso)  $xy + yz + \bar{x}z = xy + \bar{x}z$  (ou o dual  $(x + y)(y + z)(\bar{x} + z) = (x + y)(\bar{x} + z)$ )
  - $yx = zx$  e  $y\bar{x} = z\bar{x}$  implica que  $y = z$
  - $(x + y + z)(x + y) = x + y$
- Simplifique as seguintes expressões
  - $y\bar{z}(\bar{z} + \bar{z}x) + (\bar{x} + \bar{y})(\bar{x}y + \bar{x}z)$
  - $x + xyz + yz\bar{x} + wz + \bar{w}x + \bar{x}y$
- Mostre que em qualquer álgebra booleana  $\langle A, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$ ,  $x\bar{y} = 0$  se, e somente se,  $xy = x$ .

# Capítulo 5

## Expressões e Funções Booleanas

Última atualização em 12/04/2018

No capítulo anterior mostramos algumas propriedades da álgebra booleana. As letras  $x, y, z$ , ou ainda,  $a, b, c$ , foram usadas para denotar elementos da álgebra booleana. A expressão  $a + b$ , por exemplo, foi usada para indicar o elemento resultante quando o elemento  $a$  é operado com o elemento  $b$  por meio da operação  $+$ . Observe que  $a$  e  $b$  representam elementos quaisquer da álgebra booleana. Portanto, tanto  $a$  como  $b$  podem ser vistas como variáveis cujos valores podem ser quaisquer elementos da álgebra booleana. Assim, podemos considerar expressões que envolvem várias variáveis e, ao se atribuir valores (que sejam elementos da álgebra booleana em questão) a essas variáveis, podemos efetuar o “cálculo” do valor da expressão (que também será um elemento da álgebra booleana em questão).

### 5.1 Expressões Booleanas

**Variáveis e literais:** Dada uma álgebra booleana  $\langle A, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$ , uma **variável booleana** é uma variável que toma valores em  $A$ .

O **complemento** de uma variável booleana  $x$ , denotado  $\bar{x}$ , é uma variável booleana tal que  $\bar{x} = \bar{a}$  sempre que  $x = a$  para qualquer  $a \in A$ .

Um **literal** é uma variável booleana  $x$  ou o seu complemento  $\bar{x}$ .

**Expressões booleanas:** Dada uma álgebra booleana  $\langle A, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$ , uma expressão booleana em  $n$  variáveis  $x_1, x_2, \dots, x_n$  é qualquer expressão definida de acordo com as seguintes regras:

- os elementos em  $A$  são expressões booleanas;
- as variáveis  $x_1, x_2, \dots, x_n$  são expressões booleanas;
- se  $x$  e  $y$  são expressões booleanas, então são também as expressões  $(x) + (y)$ ,  $(x) \cdot (y)$  e  $\overline{(x)}$ ;
- uma expressão é booleana se e somente se pode ser obtida aplicando-se quaisquer das três regras acima um número finito de vezes.

Observe que uma expressão booleana em  $n$  variáveis  $x_1, x_2, \dots, x_n$  não necessariamente precisa conter todas as  $n$  variáveis. Parênteses podem ser removidos da expressão desde que não introduzam ambiguidades. Por exemplo, a expressão  $(x_1) + (x_2)$  pode ser escrita  $x_1 + x_2$ .

Se uma expressão pode ser derivada a partir de outra aplicando-se um número finito de vezes as propriedades da álgebra booleana, então elas são ditas **equivalentes**. O valor de expressões equivalentes, para cada atribuição de valores às variáveis booleanas, é o mesmo. Ou seja, cada expressão booleana define uma função; além disso, expressões equivalentes definem uma mesma função.

## 5.2 Funções booleanas

Conforme discutido acima, dada uma álgebra booleana  $\langle A, +, \cdot, \overline{\phantom{x}}, 0, 1 \rangle$ , uma expressão booleana em  $n$  variáveis  $x_1, x_2, \dots, x_n$  define uma **função**  $f : A^n \rightarrow A$ . O valor da função  $f$  para um elemento  $a = (a_1, a_2, \dots, a_n) \in A^n$  é calculado substituindo-se cada ocorrência de  $x_i$  na expressão por  $a_i$ , para  $i = 1, 2, \dots, n$ , e calculando-se o valor da expressão.

Note que nem todas as funções do tipo  $f : A^n \rightarrow A$  podem ser definidas por uma expressão booleana; **funções booleanas** são aquelas que podem ser definidas por uma expressão booleana.

Seja  $A(n)$  o conjunto de todas as funções booleanas em  $A$  com  $n$  variáveis e seja  $\preceq$  uma relação definida em  $A(n)$  da seguinte forma:

$$f \preceq g \iff f(\mathbf{a}) \leq g(\mathbf{a}), \forall \mathbf{a} \in A^n. \quad (5.1)$$

Seja  $(f \cdot g)(\mathbf{a}) = f(\mathbf{a}) \cdot g(\mathbf{a})$ ,  $(f + g)(\mathbf{a}) = f(\mathbf{a}) + g(\mathbf{a})$ , e  $\overline{f}(\mathbf{a}) = \overline{f(\mathbf{a})}$ ,  $\forall \mathbf{a} \in A^n$ . Fazendo  $\mathbf{0}(\mathbf{a}) = 0$  e  $\mathbf{1}(\mathbf{a}) = 1$  para todo  $\mathbf{a} \in A^n$ , pode-se mostrar que o conjunto  $(A(n), +, \cdot, \overline{\phantom{x}}, \mathbf{0}, \mathbf{1})$  é também uma álgebra booleana.

**Exemplo:** A função  $f : B^2 \rightarrow B$ , definida pela expressão  $f(x_1, x_2) = x_1 + x_2$  pode ser representada pela tabela-verdade a seguir, à esquerda. Note que ela é igual a tabela-verdade da expressão  $x_1 + \bar{x}_1 x_2$ , à direita. Logo, as expressões  $x_1 + x_2$  e  $x_1 + \bar{x}_1 x_2$  são equivalentes (ou seja, definem uma mesma função).

$x_1$	$x_2$	$x_1 + x_2$	$x_1$	$x_2$	$\bar{x}_1$	$\bar{x}_1 x_2$	$x_1 + \bar{x}_1 x_2$
0	0	0	0	0	1	0	0
0	1	1	0	1	1	1	1
1	0	1	1	0	0	0	1
1	1	1	1	1	0	0	1

Há  $2^{(2^2)} = 16$  funções de 2 variáveis para  $\langle B, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$  (ver abaixo). Como veremos mais adiante, todas elas são booleanas, no sentido de que para cada uma delas há uma expressão que a define.

$x_1$	$x_2$	$f_0$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$	$f_7$	$f_8$	$f_9$	$f_{10}$	$f_{11}$	$f_{12}$	$f_{13}$	$f_{14}$	$f_{15}$
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

As funções do tipo  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ , que são as funções booleanas com respeito à álgebra booleana  $\langle B, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$ , são também denominadas **funções lógicas** ou **funções de chaveamento**. Cabe comentar que em muitos textos na literatura o termo função booleana é usado como sinônimo de funções lógicas (i.e., funções do tipo  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ ).

**Exercício 1:** Supondo  $n$  e  $A$  finitos, quantas funções da forma  $f : \{0, 1\}^n \rightarrow \{0, 1\}$  existem? E da forma  $f : A^n \rightarrow A$  (não necessariamente booleanas) ?

**Exercício 2:** Dada a álgebra booleana  $\langle A, +, \cdot, \bar{\phantom{x}}, 0, 1 \rangle$  com  $A = \{0, 1, a, \bar{a}\}$ , construa a tabela-verdade da função correspondente à expressão  $\bar{a}x + a\bar{y}$ .

**Teorema de expansão de Boole:** Seja  $f : A^n \rightarrow A$  uma função booleana. Então,  $\forall (x_1, x_2, \dots, x_n) \in A^n$ ,

$$f(x_1, x_2, \dots, x_n) = \bar{x}_1 \cdot f(0, x_2, \dots, x_n) + x_1 \cdot f(1, x_2, \dots, x_n). \quad (5.2)$$

**Dem.**: veja, por exemplo, página 408 de R. Garnier and J. Taylor, *Discrete Mathematics for New Technology*, Adam Hilger, 1992.  $\square$

**Colorário** (dual do teorema anterior): Seja  $f : A^n \rightarrow A$  uma função booleana. Então,  $\forall(x_1, x_2, \dots, x_n) \in A^n$ ,

$$f(x_1, x_2, \dots, x_n) = [\bar{x}_1 + f(1, x_2, \dots, x_n)] \cdot [x_1 + f(0, x_2, \dots, x_n)]. \quad (5.3)$$

**Exemplo:** Seja  $A = \{0, 1, a, \bar{a}\}$ . A função  $f$  definida pela tabela a seguir é uma função booleana?

$x$	$f(x)$
0	$a$
1	1
$a$	$\bar{a}$
$\bar{a}$	1

De acordo com o teorema de expansão de Boole, sabemos que se  $f$  é uma função booleana podemos escrever  $f(x) = \bar{x}f(0) + xf(1) = \bar{x}a + x1 = \bar{x}a + x$ . Em particular, para  $x = a$  teríamos então  $f(a) = \bar{a}a + a = 0a + a = a$ . Porém, na definição de  $f$  temos  $f(a) = \bar{a}$ . Logo,  $f$  não é uma função booleana.  $\square$

**Exercício 3:** Deduza uma expressão booleana correspondente à função definida pela tabela-verdade do exercício 2 (a partir da tabela-verdade e não da expressão dada!).

**Exercício 4:** Seja  $A = \{0, 1, a, \bar{a}\}$ . Liste todas as funções booleanas em  $A$  de uma variável.

### 5.3 Somas e produtos

A partir deste ponto, iremos considerar a álgebra booleana  $B$ .

**Produto:** Um produto é uma expressão booleana que é ou um literal, ou uma conjunção<sup>1</sup> de dois ou mais literais, desde que uma variável não apareça mais de uma vez (pode aparecer apenas na forma não barrada ou apenas na forma barrada). Produtos podem ser expressos como  $p = \prod_{i=1}^n \sigma_i$ ,  $\sigma_i \in \{x_i, \bar{x}_i, ''\}$ , com  $''$  denotando o caractere vazio. Por exemplo, para  $n = 4$ ,

<sup>1</sup>Conjunção é outro nome para a operação  $\cdot$  (E lógico).

$x_1x_3$  e  $x_2\bar{x}_3\bar{x}_4$  são exemplos de produtos. As expressões  $x_1x_2\bar{x}_1$ ,  $x_1x_1$ ,  $\overline{ab}$ ,  $(a+b)c$  não são produtos.

**Mintermos:** Mintermo (ou **produto canônico**) em  $n$  variáveis  $x_1, x_2, \dots, x_n$  é uma expressão booleana formada pelo produto de cada uma das  $n$  variáveis ou dos respectivos complementos (mas não ambas). Ou seja, consiste do produto de  $n$  literais, cada um correspondendo a uma variável (se  $x_i$  está presente no produto, então  $\bar{x}_i$  não está, e vice-versa).

Denotemos  $x$  por  $x^1$  e  $\bar{x}$  por  $x^0$ . Assim, qualquer mintermo pode ser expresso por  $m_{e_1, e_2, \dots, e_n} = x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$ , com  $e_i \in \{0, 1\}$ . Por exemplo, se considerarmos  $n = 3$ , então  $m_{001} = x_1^0 x_2^0 x_3^1 = \bar{x}_1 \bar{x}_2 x_3$ . Uma vez que o conjunto de todas as sequências de  $n$  bits corresponde à representação binária dos números entre 0 e  $2^n - 1$ , um mintermo pode ser identificado por um índice decimal. A tabela 5.1 apresenta todos os mintermos em três variáveis e a notação com um índice decimal associado a cada um deles.

$e_1 e_2 e_3$	mintermo	notação
0 0 0	$\bar{x}_1 \bar{x}_2 \bar{x}_3$	$m_0$
0 0 1	$\bar{x}_1 \bar{x}_2 x_3$	$m_1$
0 1 0	$\bar{x}_1 x_2 \bar{x}_3$	$m_2$
0 1 1	$\bar{x}_1 x_2 x_3$	$m_3$
1 0 0	$x_1 \bar{x}_2 \bar{x}_3$	$m_4$
1 0 1	$x_1 \bar{x}_2 x_3$	$m_5$
1 1 0	$x_1 x_2 \bar{x}_3$	$m_6$
1 1 1	$x_1 x_2 x_3$	$m_7$

**Tabela 5.1:** Tabela de mintermos em 3 variáveis.

**Soma:** Analogamente, define-se uma **soma** como sendo ou um literal ou a disjunção de dois ou mais literais, desde que uma variável não apareça mais de uma vez (pode aparecer apenas na forma não barrada ou apenas na forma barrada).

**Maxtermo** (ou **soma canônica**) em  $n$  variáveis  $x_1, x_2, \dots, x_n$  tem definição similar ao mintermo: em vez de produto, consiste de soma de  $n$  literais, cada um correspondendo a uma variável. As expressões  $\bar{x}_1 + \bar{x}_2 + \bar{x}_3$  e  $\bar{x}_1 + x_2 + x_3$  são exemplos de maxtermos em três variáveis. A tabela 5.2 lista todos os maxtermos de 3 variáveis.

**Soma de produtos:** Dizemos que uma expressão está na forma **soma de produtos** (SOP) se ela é um produto ou se é uma disjunção de dois ou mais produtos e se nenhum par de produtos  $p$  e  $p'$  é tal que  $p \preceq p'$  (a relação  $\preceq$  é a definida pela equivalência 5.1).

**Produto de somas:** Analogamente, dizemos que uma expressão booleana está na forma **produto de somas** (POS) se ela é uma soma ou é uma conjunção de duas ou mais somas.

$e_1e_2e_3$	maxtermo	notação
0 0 0	$x_1 + x_2 + x_3$	$M_0$
0 0 1	$x_1 + x_2 + \bar{x}_3$	$M_1$
0 1 0	$x_1 + \bar{x}_2 + x_3$	$M_2$
0 1 1	$x_1 + \bar{x}_2 + \bar{x}_3$	$M_3$
1 0 0	$\bar{x}_1 + x_2 + x_3$	$M_4$
1 0 1	$\bar{x}_1 + x_2 + \bar{x}_3$	$M_5$
1 1 0	$\bar{x}_1 + \bar{x}_2 + x_3$	$M_6$
1 1 1	$\bar{x}_1 + \bar{x}_2 + \bar{x}_3$	$M_7$

**Tabela 5.2:** Tabela de maxtermos com 3 variáveis.

As expressões  $xy$ ,  $x + yz$  e  $xyw + \bar{x}z + yz$  estão na forma SOP. A expressão  $x(y + z)$  não está na forma SOP (por causa dos parênteses). A expressão  $xyzx$  não está na forma SOP (pois  $yzzx \preceq xy$ ). De fato,

$$xy + xyz = x(y + yz) = x(y \cdot 1 + yz) = x(y(1 + z)) = x(y \cdot 1) = x(y) = xy$$

**Exemplos:** Para escrever a expressão  $f(x, y, z, w) = (xz + y)(zw + \bar{w})$  na forma SOP, basta aplicarmos a distributiva para eliminar os parênteses.

$$\begin{aligned} f(x, y, z, w) &= (xz + y)(zw + \bar{w}) \\ &= (xz + y)zw + (xz + y)\bar{w} \quad (\text{distributiva}) \\ &= xzw + yzw + xz\bar{w} + y\bar{w} \quad (\text{distributiva}) \end{aligned}$$

Similarmente,

$$\begin{aligned} f(x, y, z) &= [(x + \bar{y}) + z](x + \bar{y})\bar{x} \\ &= [(x + \bar{y}) + z](x + y)\bar{x} \\ &= [(x + \bar{y}) + z](x\bar{x} + y\bar{x}) \\ &= [(x + \bar{y}) + z]y\bar{x} \\ &= xy\bar{x} + \bar{y}y\bar{x} + zy\bar{x} \\ &= 0 + 0 + zy\bar{x} \\ &= \bar{y}yz \end{aligned}$$

## 5.4 Formas canônicas

Vimos que uma mesma função pode ser expressa por diversas expressões. Assim, é natural questionarmos como verificar se duas expressões são equivalentes e também qual a melhor expressão para representar uma função. Expressões em formas canônicas são úteis pois garantem que uma função pode ser expressa unicamente nessa forma. Veremos nesta seção as formas **soma**

**canônica de produtos e produto canônico de somas.**

### 5.4.1 Soma canônica de produtos (SOP canônica)

Dizemos que uma expressão está na forma **SOP canônica** (soma canônica de produtos, ou ainda soma de mintermos) se ela é um mintermo ou se é uma disjunção de dois ou mais mintermos distintos. A seguir mostramos que qualquer função booleana pode ser expressa na forma soma de mintermos.

**Teorema:** Há  $2^n$  mintermos em  $n$  variáveis e não há dois mintermos equivalentes.

**PROVA:** Como um mintermo consiste de  $n$  literais, cada um podendo ser uma variável  $x$  ou o seu complemento  $\bar{x}$ , há no total  $2^n$  possíveis formas de se combinar os literais.

Para mostrar que não há dois mintermos equivalentes, seja  $m_{e_1, e_2, \dots, e_n} = x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$  um mintermo e considere

$$x_i = \begin{cases} 1, & \text{se } e_i = 1, \\ 0, & \text{se } e_i = 0. \end{cases}$$

Então,  $m_{e_1, e_2, \dots, e_n}(x_1, x_2, \dots, x_n) = 1$ , pois pela forma como  $x_i^{e_i}, i = 1, \dots, n$ , foram definidas, todos os literais no mintermo  $m$  tem valor 1. Isto mostra que há uma atribuição de valores às variáveis  $x_1, x_2, \dots, x_n$  que torna 1 o valor de  $m$ .

Qualquer outro mintermo  $m'$  tem pelo menos um literal  $x^{e_j}$  que é complemento do correspondente literal em  $m$ . Portanto, substituindo os valores acima das  $n$  variáveis em  $m'$ , haverá pelo menos um elemento zero no produto (devido ao literal  $x^{e_j}$ ). Isto quer dizer que  $m'$  vale zero para esses valores de  $x_1, x_2, \dots, x_n$  em particular. Portanto, para quaisquer dois mintermos, há sempre uma atribuição de valores às variáveis  $x_1, x_2, \dots, x_n$  que torna um deles 1 e o outro 0.  
□

**Teorema:** Qualquer função booleana que não seja identicamente 0 (nulo) pode ser expressa unicamente na forma **soma canônica de produtos**. Mais precisamente, se  $f$  é uma função booleana em  $n$  variáveis então a sua forma soma canônica de produtos é dada por

$$f(x_1, x_2, \dots, x_n) = \bigvee_{(e_1, e_2, \dots, e_n) \in \{0,1\}^n} f(e_1, e_2, \dots, e_n) x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$$

**PROVA:** Esta igualdade decorre do teorema de expansão de Boole, e a unicidade de representação está relacionada ao fato de mintermos serem distintos dois a dois. Uma demonstração completa

pode ser encontrada em R. Garnier and J. Taylor, *Discrete Mathematics for New Technology*, Adam Hilger, 1992, página 408.  $\square$

**Exemplo:** Para ilustrar o teorema acima, considere uma função booleana  $f$  em 3 variáveis,  $x_1$ ,  $x_2$  e  $x_3$ . Ao aplicarmos o teorema de expansão de Boole recursivamente, obtemos a seguinte expansão:

$$\begin{aligned} f(x_1, x_2, x_3) &= \bar{x}_1 f(0, x_2, x_3) + x_1 f(1, x_2, x_3) \\ &= \bar{x}_1 [\bar{x}_2 f(0, 0, x_3) + x_2 f(0, 1, x_3)] + x_1 [\bar{x}_2 f(1, 0, x_3) + x_2 f(1, 1, x_3)] \\ &= \bar{x}_1 \bar{x}_2 \bar{x}_3 f(0, 0, 0) + \bar{x}_1 \bar{x}_2 x_3 f(0, 0, 1) + \bar{x}_1 x_2 \bar{x}_3 f(0, 1, 0) + \bar{x}_1 x_2 x_3 f(0, 1, 1) + \\ &\quad x_1 \bar{x}_2 \bar{x}_3 f(1, 0, 0) + x_1 \bar{x}_2 x_3 f(1, 0, 1) + x_1 x_2 \bar{x}_3 f(1, 1, 0) + x_1 x_2 x_3 f(1, 1, 1) \end{aligned}$$

Após substituirmos os valores de  $f$  nos pontos  $\{0, 1\}^3$  na expressão acima, restarão apenas os produtos canônicos (mintermos)  $m_i$  tais que  $m_i \preceq f$ .

**Teorema:** Uma função  $f : B^n \rightarrow B$  é booleana se e somente se ela pode ser expressa na forma soma canônica de produtos, ou seja,  $\forall (x_1, x_2, \dots, x_n) \in B^n$ ,

$$f(x_1, x_2, \dots, x_n) = \bigvee_{(e_1, e_2, \dots, e_n) \in \{0, 1\}^n} f(e_1, e_2, \dots, e_n) x_1^{e_1} x_2^{e_2} \dots x_n^{e_n} \quad (5.4)$$

**Dem.:** Se  $f$  é uma função booleana, então ela pode ser expressa na forma soma canônica de produtos conforme teorema anterior. Por outro lado, suponha que  $f$  pode ser expressa na forma da equação 5.4. Claramente, a expressão é uma expressão booleana e portanto  $f$  é uma função booleana.  $\square$

Observe que, para uma dada função booleana, a representação na forma soma canônica de produtos é única, a menos da ordem dos mintermos. Por outro lado, podem existir duas funções distintas cujas respectivas formas soma canônica de produtos é idêntica? A resposta é não, pois uma expressão define uma única função.

**Exemplos:** Para escrever a função  $f(x_1, x_2) = x_1 + x_2$  na forma SOP canônica, podemos aplicar o teorema de expansão de Boole:

$$f(x_1, x_2) = f(0, 0) \bar{x}_1 \bar{x}_2 + f(0, 1) \bar{x}_1 x_2 + f(1, 0) x_1 \bar{x}_2 + f(1, 1) x_1 x_2$$

e em seguida calcular o valor de  $f$  para todos os elementos  $\mathbf{e} \in \{0, 1\}^2$ . Temos,  $f(0, 0) = 0$  e  $f(0, 1) = f(1, 0) = f(1, 1) = 1$  e, portanto,

$$f(x_1, x_2) = 0 \cdot \bar{x}_1 \bar{x}_2 + 1 \cdot \bar{x}_1 x_2 + 1 \cdot x_1 \bar{x}_2 + 1 \cdot x_1 x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2 + x_1 x_2 \quad (5.5)$$

No caso de funções do tipo  $f : B^n \rightarrow B$ , todas podem ser expressas por expressões booleanas. Uma forma prática de gerarmos a forma canônica a partir de uma tabela-verdade está mostrada a seguir. Seja a tabela-verdade (**OBS.:** Lembrem-se de sempre escrever as entradas da tabela verdade em ordem lexicográfica) a seguir:

Entrada			Saída
$x_1$	$x_2$	$x_3$	$f(x_1, x_2, x_3)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

A função definida na tabela-verdade acima toma valor 1 para as entradas 000, 001, 010, 011 e 111. Para cada entrada, conforme já vimos, há apenas um produto canônico que toma valor 1. Por exemplo, para a entrada 010 o produto canônico que toma valor 1 é o produto  $\bar{x}_1 x_2 \bar{x}_3$  (pois,  $\bar{x}_1 x_2 \bar{x}_3 = \bar{0} \bar{1} \bar{0} = 1 \cdot 1 \cdot 1 = 1$ ). Para qualquer outra entrada, esse produto toma valor 0 e é por isso que podemos associar um único produto canônico para cada entrada. Portanto, uma função pode ser escrita como a soma dos produtos associados às entradas para as quais ela toma valor 1. Em particular, a função definida pela tabela acima pode ser escrita como:

$$f(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3 \quad (5.6)$$

Usando-se o índice decimal associado a cada produto canônico, podemos escrever a expressão da função  $f$  de forma mais compacta como:

$$f(x_1, x_2, x_3) = m_0 + m_1 + m_2 + m_3 + m_7 = \sum m(0, 1, 2, 3, 7)$$

Pode-se também obter a forma canônica de uma expressão qualquer, por meio de manipulações algébricas (eliminar os parênteses e complementos de subexpressões e em seguida “introduzir”, em cada produto, as variáveis que não aparecem). Por exemplo, no caso da expressão  $f(x, y, z, w) = (xz + y)(zw + \bar{w})$  podemos fazer:

$$\begin{aligned} f(x, y, z, w) &= (xz + y)zw + (xz + y)\bar{w} \\ &= xzw + yzw + xz\bar{w} + y\bar{w} \\ &= xzw(y + \bar{y}) + (x + \bar{x})yzw + x(y + \bar{y})z\bar{w} + (x + \bar{x})y(z + \bar{z})\bar{w} \\ &= xyzw + x\bar{y}zw + xyzw + \bar{x}yzw + xyz\bar{w} + x\bar{y}z\bar{w} + xy(z + \bar{z})\bar{w} + \bar{x}y(z + \bar{z})\bar{w} \\ &= xyzw + x\bar{y}zw + \bar{x}yzw + xyz\bar{w} + x\bar{y}z\bar{w} + xyz\bar{w} + xy\bar{z}\bar{w} + \bar{x}yz\bar{w} + \bar{x}y\bar{z}\bar{w} \\ &= xyzw + xyz\bar{w} + xy\bar{z}\bar{w} + x\bar{y}zw + x\bar{y}z\bar{w} + \bar{x}yzw + \bar{x}yz\bar{w} + \bar{x}y\bar{z}\bar{w} \end{aligned}$$

Similarmente, no caso da função  $f(x_1, x_2) = x_1 + x_2$  podíamos ter feito

$$\begin{aligned} f(x_1, x_2) &= x_1 + x_2 = x_1(x_2 + \bar{x}_2) + (x_1 + \bar{x}_1)x_2 \\ &= x_1 x_2 + x_1 \bar{x}_2 + x_1 x_2 + \bar{x}_1 x_2 \\ &= \bar{x}_1 x_2 + x_1 \bar{x}_2 + x_1 x_2 \end{aligned}$$

### 5.4.2 Produto canônico de somas (POS canônica)

Todos os conceitos e resultados acima com respeito a expressões do tipo soma de produtos podem também ser definidos com respeito a expressões do tipo produto de somas.

**Teorema:** Há  $2^n$  maxtermos e não há dois maxtermos equivalentes.

A demonstração é similar ao caso dos mintermos.

Dizemos que uma expressão booleana está na forma **produto canônico de somas** (POS canônica ou produto de maxtermos) se ela é um maxtermo ou é uma conjunção de dois ou mais maxtermos distintos.

**Teorema:** Qualquer função booleana que não seja identicamente 1 pode ser expressa unicamente na forma **produto canônico de somas**.

**PROVA:** A demonstração desse teorema é dual à demonstração do teorema relativo à soma canônica de produtos. Em particular, deve-se considerar o teorema de expansão de Boole dual, visto anteriormente.  $\square$

**Exemplo:** A expressão  $x + z + \bar{y}\bar{w}$  na forma POS canônica pode ser calculada como segue:

$$\begin{aligned} f(x, y, z, w) &= x + z + \bar{y}\bar{w} \\ &= x + (z + \bar{y}\bar{w}) \\ &= x + (z + \bar{y})(z + \bar{w}) \\ &= (x + z + \bar{y})(x + z + \bar{w}) \\ &= (x + \bar{y} + z + w\bar{w})(x + y\bar{y} + z + \bar{w}) \\ &= (x + \bar{y} + z + w)(x + \bar{y} + z + \bar{w})(x + y + z + \bar{w})(x + \bar{y} + z + \bar{w}) \\ &= (x + \bar{y} + z + w)(x + \bar{y} + z + \bar{w})(x + y + z + \bar{w}) \end{aligned}$$

**Exemplo:** Seja a função que toma valor zero para as entradas 100, 101 e 110 e 1 para as demais (esta função foi apresentada acima). A soma  $\bar{x}_1 + x_2 + x_3$  toma valor 0 para a entrada 100 e

valor 1 para as demais entradas. Como um produto de somas toma valor 0 se ao menos um dos termos soma tomar valor 0, podemos escrever a função como o produto de somas canônicas associadas às entradas para as quais a função toma valor 0. No caso do exemplo em questão,

$$f(x_1, x_2, x_3) = (\bar{x}_1 + x_2 + x_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + x_3) \quad (5.7)$$

Essa expressão está na forma POS canônica, que em sua forma compacta, é:

$$f(x_1, x_2, x_3) = M_4 \cdot M_5 \cdot M_6 = \prod M(4, 5, 6)$$

### Observações:

- Em vez de **produto**, alguns autores utilizam também os nomes **termo produto**, **produto fundamental**, **conjunção fundamental** ou **produto normal**.
- Em vez de **soma de produtos**, utilizam-se também os nomes **soma de produtos normais** e **forma normal disjuntiva**.
- Em vez de **soma canônica de produtos** (SOP canônica), utilizam-se também os nomes **soma padrão de produtos**, **forma normal disjuntiva completa** ou **forma min-termo**. Note, porém, que alguns autores usam o nome **forma normal disjuntiva** em vez de **forma normal disjuntiva completa**.
- Nós usaremos **soma de produtos** e **soma canônica de produtos** (ou **soma de min-termos**).

### Exercícios:

1. Liste todos os mintermos em 3 variáveis.
2. Escreva  $f(a, b, c, d, e) = (\bar{a}\bar{c} + \bar{d})(\bar{b} + c\bar{e})$  na forma SOP.
3. Escreva  $f(a, b, c, d) = (a + b)c\bar{d} + (a + b)\bar{c}d$  na forma SOP canônica.
4. Escreva  $f(x, y, z, w) = x + z + \bar{y}\bar{w}$  na forma SOP canônica. Existe relação entre a forma SOP canônica e a forma POS canônica? Qual?
5. Ache a expressão na forma SOP canônica que define a função dada pela tabela-verdade abaixo. Você consegue simplificar esta expressão e obter uma outra equivalente e mais curta?

$x$	$y$	$z$	$f(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

6. Prove que qualquer função booleana que não seja identicamente 0 (nulo) pode ser expressa na forma SOP canônica.
7. Prove que a forma SOP canônica de qualquer função booleana que não seja identicamente 0 (nulo) é única, a menos da ordem dos produtos canônicos.

# Capítulo 6

## Minimização de funções booleanas

Última atualização em 21/04/2018

Nos capítulos anteriores vimos que uma mesma função pode ser expressa por mais de uma expressão. Dependendo do contexto no qual essas expressões são consideradas, pode-se desejar encontrar, dentre todas as expressões que representam uma mesma função booleana, aquela que satisfaz algum critério. Por exemplo, pode ser do interesse obter uma expressão “mais curta”, ou então, uma expressão que não envolve um termo produto com mais de um determinado número de literais.

Uma das simplificações bastante estudadas no contexto de circuitos lógicos é a **minimização lógica dois-níveis**, na qual o objetivo é encontrar uma expressão na forma SOP (ou POS), envolvendo o menor número possível de termos produto (ou soma) e, em cada produto (ou soma), o menor número possível de literais. Ao se considerar a implementação da expressão SOP como circuito digital, cada termo produto corresponde a uma porta E e o número de literais em um produto corresponde ao número de entradas na correspondente porta E.

### 6.1 Formas minimais

**Definição:** Uma expressão lógica escrita na forma soma de produtos é **minimal** se

1. não existe nenhuma outra expressão equivalente<sup>1</sup> na forma soma de produtos com um número menor de termos, e

---

<sup>1</sup>Duas expressões são equivalentes se definem uma mesma função.

2. não existe nenhuma outra expressão equivalente na forma soma de produtos com igual número de termos mas com menor número de literais.

Denominaremos tal expressão de **SOP minimal<sup>2</sup>**. Note que ao se remover, de uma forma SOP minimal, um produto ou um literal de qualquer um dos produtos, a expressão resultante não mais representa a mesma função.

O problema pode ser analogamente definido para a forma POS. O problema de encontrar a forma SOP ou POS minimal de uma função é denominada **minimização lógica dois-níveis** devido ao fato de funções na forma SOP ou POS poderem ser realizadas em circuitos de dois níveis (isto é, no caso da forma SOP, o primeiro nível é formado pelas portas E, uma para cada produto, e o segundo nível consiste de uma porta OU que recebe como entradas as saídas das portas E do nível anterior).

Algoritmos de minimização lógica dois-níveis assumem, em geral, que a expressão está na forma SOP. As técnicas mais clássicas como Mapas de Karnaugh e minimização de Quine-McCluskey assumem que a expressão inicial está na forma SOP canônica.

**Exemplo:** Sejam três variáveis  $a$ ,  $b$  e  $c$ . Dados os produtos  $abc$  e  $\bar{a}bc$ , seja a disjunção (soma)  $abc + \bar{a}bc$ . Observe que podemos usar a propriedade distributiva e escrever:  $abc + \bar{a}bc = (a + \bar{a})bc = bc$  (já que  $a + \bar{a} = 1$  sempre e, portanto,  $(a + \bar{a})bc = 1 \cdot bc = bc$ ). O termo resultante  $bc$  é também um produto, porém sem a variável  $a$ . Note que o produto  $bc$  toma valor 1 para as entradas 011 e 111; o valor da variável  $a$  não afeta o valor desse produto. Veja uma compilação na tabela a seguir:

Expressão	Entradas para as quais a expressão toma valor 1
$\bar{a}bc$	{011}
$abc$	{111}
$\bar{a}bc + abc = bc$	{011, 111}

Em termos de circuitos, a simplificação acima significa que duas portas lógicas E de três entradas cada podem ser substituídas por uma porta E de duas entradas! Além disso, não será mais necessária uma porta OU.

**Breve pausa:** Podemos definir uma relação  $\leq$  em  $B = \{0, 1\}$  dizendo que  $0 \leq 0$ ,  $0 \leq 1$  e  $1 \leq 1$ . Essa relação pode ser estendida para  $B^n$  fazendo  $a_1 a_2 \cdots a_n \leq b_1 b_2 \cdots b_n \iff a_i \leq b_i, \forall i = 1, 2, \dots, n$ , para quaisquer dois elementos  $a_1 a_2 \cdots a_n$  e  $b_1 b_2 \cdots b_n$  em  $B^n$ . Por exemplo, em  $B^3$ ,  $010 \leq 011$  mas  $010 \not\leq 101$ .

Com isso podemos definir um intervalo  $[a_1 a_2 \cdots a_n, b_1 b_2 \cdots b_n] = \{x_1 x_2 \cdots x_n : a_1 a_2 \cdots a_n \leq x_1 x_2 \cdots x_n \leq b_1 b_2 \cdots b_n\}$ , com extremo inferior  $a_1 a_2 \cdots a_n$  e extremo superior  $b_1 b_2 \cdots b_n$ . Por exemplo,  $[010, 111] = \{010, 011, 110, 111\}$ .

<sup>2</sup>SOP é contração de *Sum of products*

Um produto pode ser associado a um intervalo. No caso do produto  $bc$ , as entradas para as quais o produto toma valor 1 é dado pelo conjunto  $\{011, 111\}$ . Esse conjunto tem um extremo inferior, 011, e um extremo superior, 111, definindo o intervalo  $[011, 111]$ . Um intervalo pode ser representado compactamente trocando-se as coordenadas não fixas por  $X$ . Assim  $X11$  corresponde ao intervalo  $[011, 111]$ . Se o produto considerado fosse  $a$ , teríamos que ele toma valor 1 para as entradas  $\{100, 101, 110, 111\}$  que pode ser entendido como o intervalo  $[100, 111]$ . Em notação compacta, esse intervalo pode ser denotado por  $1XX$ . Similarmente, um intervalo pode ser associado a um produto. No caso de três variáveis  $a, b$  e  $c$ , o intervalo  $[000, 100] = \{000, 100\}$  (ou simplesmente  $X0X$ ) correponde ao produto  $\bar{b}\bar{c}$ . O intervalo  $X0X = [000, 101] = \{000, 001, 100, 101\}$  corresponde ao produto  $\bar{b}$ . Um intervalo contém necessariamente  $2^k$  elementos, onde  $0 \leq k \leq n$ .

Esse conceitos/terminologias estão resumidos no quadro a seguir:

Produto	elementos cobertos	intervalo	notação compacta (cubo/intervalo)	dimensão ( $k$ )	tamanho ( $2^k$ )
$ab$	$\{110, 111\}$	$[110, 111]$	$11X$	1	$2^1 = 2$
$c$	$\{001, 011, 101, 111\}$	$[001, 111]$	$XX1$	2	$2^2 = 4$

**Observação:** Daqui em diante utilizaremos equivalentemente os termos **produto**, **cubo** ou **intervalo** quando nos referirmos a um produto.

### 6.1.1 Simplificação algébrica

Usando regras algébricas, como por exemplo em  $a b c + \bar{a} b c = (a + \bar{a}) b c = b c$ , podemos realizar a simplificação de uma expressão. A função  $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$  pode ser simplificada como segue:

$$\begin{aligned}
 f(x_1, x_2, x_3) &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3 \\
 &= \bar{x}_1 \bar{x}_2 (\bar{x}_3 + x_3) + \bar{x}_1 x_2 (\bar{x}_3 + x_3) + x_1 x_2 x_3 \\
 &= \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2 + x_1 x_2 x_3 \\
 &= \bar{x}_1 (\bar{x}_2 + x_2) + x_1 x_2 x_3 \\
 &= \bar{x}_1 + x_1 x_2 x_3 \\
 &= \bar{x}_1 + x_2 x_3
 \end{aligned}$$

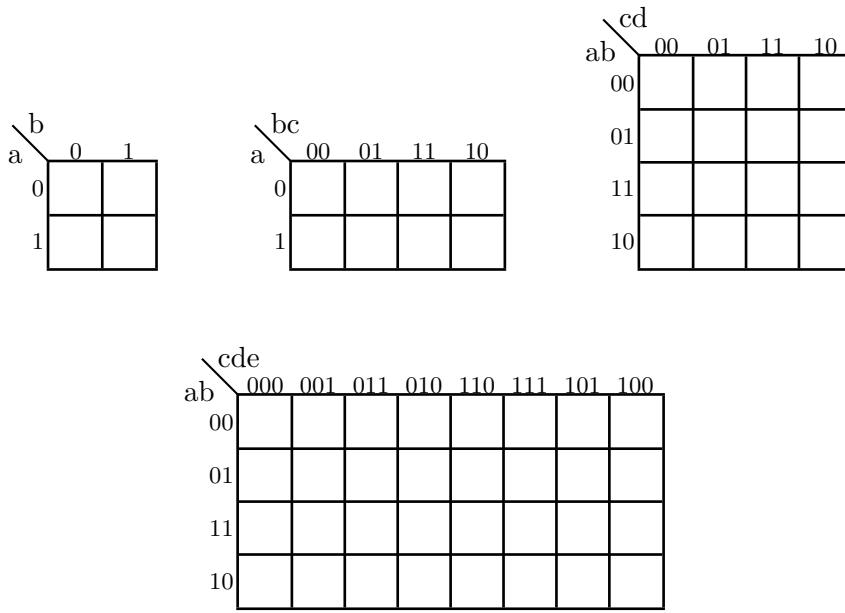
A simplificação algébrica, porém, pode ser complicada pois requer o conhecimento das regras e, além disso, dependendo da ordem na qual as regras são aplicadas pode ser necessário um grande número de passos para se chegar à simplificação (sem contar que às vezes podemos, depois de vários passos, voltar à situação inicial). Se a expressão envolver muitas variáveis, fazer a

simplificação na mão traz adicionalmente uma grande chance de se cometer erros bobos (trocar o nome de uma variável sem querer, esquecer o barra sobre uma variável de um passo para o outro, etc). Além disso, pode não ser trivial verificar que a forma mais simplificada possível foi obtida (relembre o teorema do consenso, que afirma que,  $x y + y z + \bar{x} z = x y + \bar{x} z$ ; não é trivial perceber que a expressão da esquerda é igual à expressão da direita).

## 6.2 Mapas de Karnaugh

Mapas de Karnaugh são diagramas que são utilizados para auxiliar o processo de minimização lógica dois-níveis, explorando uma representação gráfica adequada. No entanto, seu uso restringe-se à minimização de funções com não mais que 6 variáveis. Apesar disso, é um recurso pedagógico interessante e será explorado a seguir.

A estrutura de mapas de Karnaugh de 2 a 5 variáveis é exibida a seguir:



Cada célula de um mapa de  $n$  variáveis corresponde a um elemento de  $\{0, 1\}^n$  (portanto o mapa contém  $2^n$  células). A concatenação do rótulo da linha com o rótulo da coluna de uma célula dá o elemento correspondente àquela célula, conforme mostrado no mapa à esquerda na figura 6.1. O mapa à direita mostra em cada célula o valor decimal das respectivas entradas.

Observe que o rótulo está disposto em uma sequência não-usual. Por exemplo, para duas variáveis, a sequência natural seria 00, 01, 10, 11. Porém, a sequência utilizada é 00, 01, 11, 10, que possui a característica de dois elementos adjacentes (na sequência) diferirem em apenas 1 bit (note que essa propriedade vale também em relação aos extremos esquerdo e direito do mapa).

		$bc$						$bc$			
		00	01	11	10			00	01	11	10
$a$	0	000	001	011	010	$a$	0	0	1	3	2
	1	100	101	111	110		1	4	5	7	6

**Figura 6.1:** As entradas correspondentes a cada célula do mapa de Karnaugh de 3 variáveis em notação binária e decimal, respectivamente.

e, similarmente, aos extremos superior e inferior). Esse tipo de sequência é conhecido como *gray code* e pode ser generalizado para  $n$  bits.

### 6.2.1 Minimização usando mapas de Karnaugh

Mostraremos a seguir como realizar a minimização de uma função de três variáveis, utilizando o mapa de Karnaugh. Utilizaremos a mesma função  $f$  acima que foi simplificada algebraicamente, a qual reescrevemos aqui:

$$f(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3 = \bar{x}_1 + x_2 x_3$$

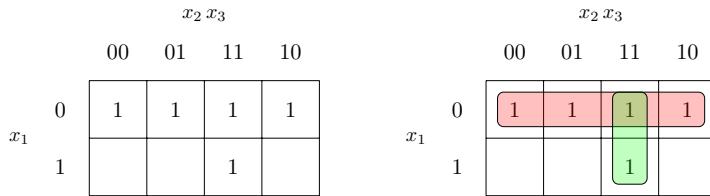
Aparentemente a expressão simplificada acima é minimal. Para utilizarmos o mapa de Karnaugh, precisamos primeiramente transformar os mintermos da função para a notação cúbica. Assim,

Mintermo	notação cúbica
$\bar{x}_1 \bar{x}_2 \bar{x}_3$	000
$\bar{x}_1 \bar{x}_2 x_3$	001
$\bar{x}_1 x_2 \bar{x}_3$	010
$\bar{x}_1 x_2 x_3$	011
$x_1 x_2 x_3$	111

Lembre-se que os mintermos da função são os produtos canônicos associados às entradas para as quais a função toma valor 1.

Em seguida, as células do mapa correspondentes a esses mintermos devem ser marcadas com 1, conforme mostrado no mapa da esquerda na figura 6.2.

O processo de minimização consiste, então, em procurar, para cada 1 no mapa, o maior agrupamento retangular formado por 1's adjacentes ao 1 em questão. Os agrupamentos devem

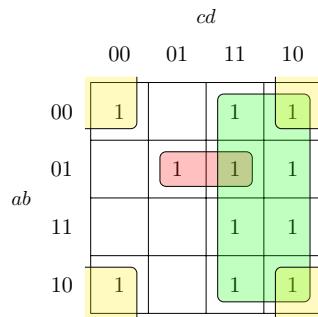


**Figura 6.2:** Exemplo do uso do mapa de Karnaugh: minimização da função  $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$ . À esquerda, as células correspondentes às entradas nas quais  $f$  toma valor 1 são preenchidas com 1. À direita, o resultado da minimização: em verde o intervalo X11 que corresponde ao produto  $x_2 x_3$  e em vermelho o intervalo 0XX que corresponde ao produto  $\bar{x}_1$ .

sempre conter  $2^k$  elementos ( $k \geq 1$ ) e eles são chamados **cubos maximais**. No exemplo da figura 6.2, o maior agrupamento com tais características que cobre 000 é o cubo 0XX. De fato, o cubo 0XX cobre os mintermos 000, 001, 010 e 011. Algebricamente, esse cubo é equivalente a  $\bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3$  que por sua vez é equivalente a  $\bar{x}_1 \bar{x}_2 (\bar{x}_3 + x_3) + \bar{x}_1 x_2 (\bar{x}_3 + x_3) = \bar{x}_1 \bar{x}_2 + \bar{x}_1 x_2 = \bar{x}_1 (\bar{x}_2 + x_2) = \bar{x}_1$ . Isto é, o processo de agrupar 1s, do ponto de vista algébrico, corresponde ao processo de eliminar variáveis substituindo dois ou mais produtos por um. Observe que a expressão  $\bar{x}_1$  corresponde ao cubo 0XX.

Voltando ao mapa da figura 6.2, o cubo 0XX não cobre o elemento 111. Assim, tomamos também o maior cubo que cobre 111, que no caso é o cubo X11. Depois desse procedimento, todos os mintermos da função encontram-se cobertos por algum cubo. Assim, podemos dizer que uma solução SOP minimal corresponde aos cubos 0XX e X11. O produto correspondente ao cubo 0XX é  $\bar{x}_1$  e o correspondente a X11 é  $x_2 x_3$ . Portanto, uma forma SOP minimal é  $f(x_1, x_2, x_3) = \bar{x}_1 + x_2 x_3$ .

**Exemplo:** Minimize a função  $f(a, b, c, d) = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$ . A resposta é  $f(a, b, c, d) = c + \bar{a} b d + \bar{b} \bar{d}$ . Veja o mapa da figura 6.3. Note que os elementos dos quatro cantos formam um cubo.

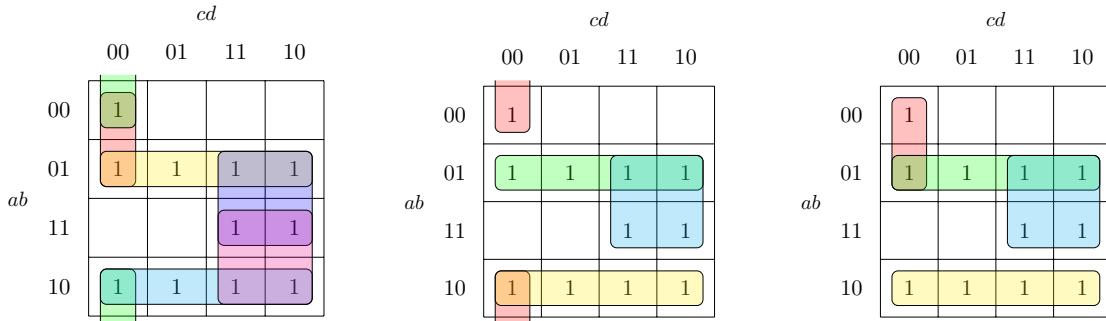


**Figura 6.3:** Minimização da função  $f(a, b, c, d) = \sum m(0, 2, 3, 5, 6, 7, 8, 10, 11, 14, 15)$ . Em verde temos XX1X (c), em vermelho 01X1 (  $\bar{a} b d$  ) e em amarelo X0X0 (  $\bar{b} \bar{d}$  ).

**Exemplo:** Minimize a função  $f(a, b, c, d) = \sum m(0, 4, 5, 6, 7, 8, 9, 10, 11, 14, 15)$ . Neste caso, há mais de uma solução. O mapa mais à esquerda da figura 6.4 mostra todos os cubos maximais de  $f$ . As possíveis soluções são listadas a seguir, duas das quais são ilustradas na figura 6.4 (os dois mapas da direita).

$$\begin{aligned}f(a, b, c, d) &= \bar{a}b + a\bar{b} + \bar{a}\bar{c}\bar{d} + bc \\f(a, b, c, d) &= \bar{a}b + a\bar{b} + \bar{a}\bar{c}\bar{d} + ac\end{aligned}$$

$$\begin{aligned}f(a, b, c, d) &= \bar{a}b + a\bar{b} + \bar{b}\bar{c}\bar{d} + bc \\f(a, b, c, d) &= \bar{a}b + a\bar{b} + \bar{b}\bar{c}\bar{d} + ac\end{aligned}$$



**Figura 6.4:** Minimização da função  $f(a, b, c) = \sum m(0, 4, 5, 6, 7, 8, 9, 10, 11, 14, 15)$ . Esquerda: todos os cubos maximais (verde=X000, salmão=0X00, amarelo=01XX, azul=10XX, lilás=X11X, rosa=1X1X). Centro: uma solução. Direita: outra solução. Note que há mais duas (e também que as cores não se mantêm do primeiro mapa para os demais ...)

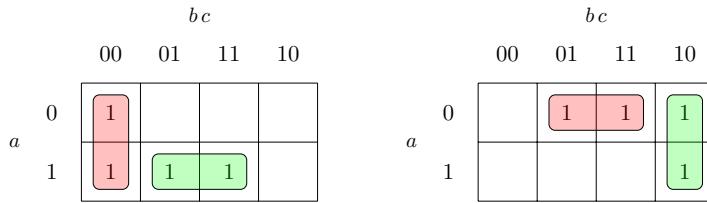
Este exemplo mostra que a solução não é única (ou seja, podem existir mais de uma forma SOP minimal de mesmo custo) e que nem todos os cubos maximais<sup>3</sup> fazem parte da forma SOP minimal.

**Mapa de Karnaugh para encontrar a forma POS minimal:** o mapa de Karnaugh pode ser utilizado também para encontrar a forma POS (produto de somas) minimal de uma função booleana. Considere a função  $f(a, b, c) = \sum m(0, 4, 5, 7)$ . A minimização SOP de  $f$  por mapa de Karnaugh é mostrada na figura 6.5, à esquerda. À direita é mostrada a minimização de  $\bar{f}(a, b, c) = \sum m(1, 2, 3, 6)$

A partir do mapa da direita, temos que  $\bar{f}(a, b, c) = b\bar{c} + \bar{a}c$ . Usando o fato de que  $f = \bar{\bar{f}}$ , segue que  $f = \overline{b\bar{c} + \bar{a}c} = (\bar{b}\bar{c})(\bar{a}c) = (\bar{b} + c)(a + \bar{c})$ .

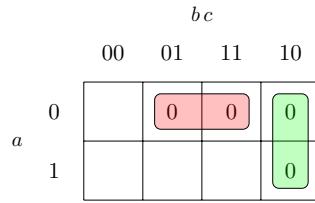
Tudo isto pode ser diretamente realizado no mapa de Karnaugh conforme mostrado na figura 6.6. Em vez de marcar os mintermos de  $f$  no mapa, marcamos os maxtermos de  $f$  (ou seja, os mintermos de  $\bar{f}$  ou, ainda, os zeros da função  $f$ ). Aplica-se o processo de encontrar os cubos maximais em relação aos zeros, da mesma forma que fizemos com os 1s no caso da forma SOP.

<sup>3</sup>Cubo maximal é o equivalente a um produto que não pode mais ser simplificado, isto é, se eliminarmos mais uma variável do produto, a função resultante passaria a tomar valor 1 em uma entrada na qual ela vale 0. No caso do cubo ou intervalo no mapa de Karnaugh, maximal significa que se estendermos o cubo ele passaria a cobrir um zero, algo que não queremos.



**Figura 6.5:** À esquerda, minimização da função  $f(a, b, c) = \sum m(0, 4, 5, 7)$  (em vermelho X00 correspondente a  $\bar{b}\bar{c}$  e, em verde, 1X1 correspondente a  $a\bar{c}$ ). À direita, minimização da função  $\bar{f}(a, b, c) = \sum m(1, 2, 3, 6)$

Para escrever a função na forma POS minimal, precisamos escrever, para cada cubo maximal encontrado, a soma que toma valor zero para todos os elementos do cubo. Por exemplo, no caso do cubo 0X1 queremos a soma que toma valor 0 para as entradas 001 e 011 (e apenas para essas duas entradas). Sabemos que o termo produto correspondente ao cubo 0X1 é  $\bar{a}c$  e ele toma valor 1 para as entradas 001 e 011. Logo, o que queremos é  $\bar{a}\bar{c}$  que é igual à soma  $a + \bar{c}$ . Fazendo o mesmo para o segundo cubo, temos então que a forma POS minimal é  $f(a, b, c) = (a + \bar{c})(\bar{b} + c)$ .



**Figura 6.6:** Minimização na forma POS da função  $f(a, b, c) = \prod M(1, 2, 3, 6)$  (em vermelho o cubo 0X1 que corresponde à soma  $a + \bar{c}$  e, em verde, o cubo X10 que corresponde à soma  $\bar{b} + c$ ).

### 6.2.2 Minimização na presença de don't cares

Em algumas situações, o valor de uma função para algumas entradas não é relevante (tipicamente porque tais entradas nunca ocorrerão na prática). Em tais situações, tanto faz se a função toma valor 0 ou 1 nessas entradas, que serão denominadas de **don't cares**.

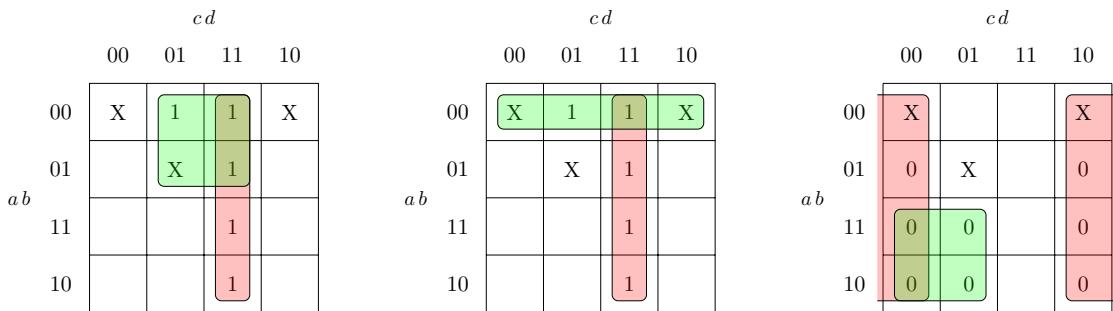
Para minimizar uma função incompletamente especificada, os *don't cares* podem ser utilizados da forma mais conveniente possível. Isto é, se a minimização é na forma SOP, os *don't cares* podem ser usados para formar cubos maximais junto com os 1's da função. Analogamente, se a minimização é na forma POS, eles podem ser usados para formar cubos maximais junto com os 0's da função. Podem também ser completamente ignorados.

Por exemplo, seja  $f(a, b, c, d) = \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$ . Esta função  $f$  vale 1 para 0001, 0011, 0111, 1011, e 1111; vale 0 para 0100, 0110, 1000, 1001, 1010, 1100, 1101 e 1110; para 0000,

0010 e 0101 não importa o valor de  $f$ . No mapa, preenchemos *don't cares* com X (veja abaixo).

		$cd$			
		00	01	11	10
$ab$	00	X	1	1	X
	01	0	X	1	0
	11	0	0	1	0
	10	0	0	1	0

Para minimizar na forma SOP, consideramos os 1 e os X, conforme os dois mapas da esquerda abaixo. Esses dois mapas ilustram duas possíveis soluções de mesmo custo. O primeiro corresponde a  $\bar{a}d + cd$  e o segundo a  $\bar{a}\bar{b} + cd$ . Já para minimizar na forma POS, consideramos os 0 e X (mapa mais a direita). O resultado na forma POS é  $d(\bar{a} + c)$ . Note que em ambos os casos os *don't cares* ajudam na minimização. Ao compararmos a forma SOP e POS minimais, concluímos que a POS minimal é vantajosa pois requer uma porta OU e uma porta E, enquanto a forma SOP requer duas portas E e uma porta OU.



### 6.3 Minimização de múltiplas funções

Sejam as funções  $f_1$  e  $f_2$  dadas pelas seguintes tabelas:

$x_1 x_2 x_3$	$f_1(x_1, x_2, x_3)$	$x_1 x_2 x_3$	$f_2(x_1, x_2, x_3)$
000	1	000	0
001	1	001	0
010	0	010	0
011	0	011	0
100	0	100	0
101	1	101	1
110	0	110	1
111	0	111	1

Minimizando-se individualmente estas funções, temos

$$f_1(x_1, x_2, x_3) = \bar{x}_1 \bar{x}_2 + \bar{x}_2 x_3$$

$$f_2(x_1, x_2, x_3) = x_1 x_3 + x_1 x_2$$

Para implementá-las, são necessárias 4 portas E.

Note, porém, que podemos escrever

$$f_1(x_1 x_2 x_3) = \bar{x}_1 \bar{x}_2 + \bar{x}_2 x_3 = \bar{x}_1 \bar{x}_2 + x_1 \bar{x}_2 x_3$$

$$f_2(x_1 x_2 x_3) = x_1 x_3 + x_1 x_2 = x_1 x_2 + x_1 \bar{x}_2 x_3$$

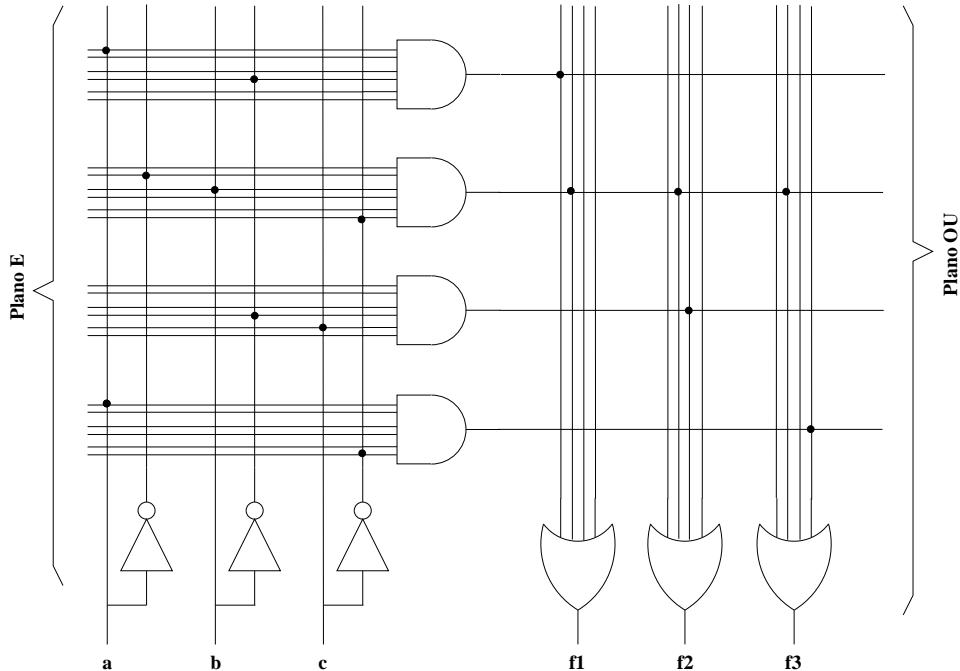
Neste caso, nenhuma das expressões está na forma minimal, mas há um produto comum às duas funções. Logo para implementá-las são necessárias 3 portas E, pois uma das portas E pode ser compartilhada pelas duas funções.

Este exemplo mostra que minimizar individualmente as funções não necessariamente representa a melhor solução para a minimização de múltiplas funções. Na minimização de múltiplas funções deve-se buscar também a maximização de compartilhamento.

## 6.4 PLA

A minimização lógica dois-níveis ganhou impulso na década de 1980 devido aos dispositivos conhecidos como **PLA** (*Programmable Logic Arrays*). Eles consistem de um conjunto de entradas, com uma malha programável de conexões para um conjunto de portas E, e uma malha programável de conexões entre as saídas das portas E para um conjunto de portas OU. Por malha programável entende-se que os cruzamentos podem ser conectados (programados) para conduzir o sinal. No estado inicial, nenhum cruzamento está conectado nas malhas de um PLA.

A figura 6.7 mostra um modelo lógico básico de um PLA típico, com 3 variáveis de entrada e três saídas. Os círculos (pontos pretos) sobre o cruzamento das linhas indicam onde há conexão. No exemplo, as portas lógicas E realizam, respectivamente de cima para baixo, as funções (produtos)  $a\bar{b}$ ,  $\bar{a}b\bar{c}$ ,  $\bar{b}c$  e  $a\bar{c}$ ; as portas lógicas OU realizam, respectivamente da esquerda para a direita, as funções  $f_1(a, b, c) = \bar{a}b\bar{c} + a\bar{b}$ ,  $f_2(a, b, c) = \bar{a}b\bar{c} + \bar{b}c$  e  $f_3(a, b, c) = \bar{a}b\bar{c} + a\bar{c}$ .



**Figura 6.7:** Esquema lógico de um PLA.

Para não sobrecarregar o diagrama, em geral desenha-se de forma simplificada como o mostrado na figura 6.8.

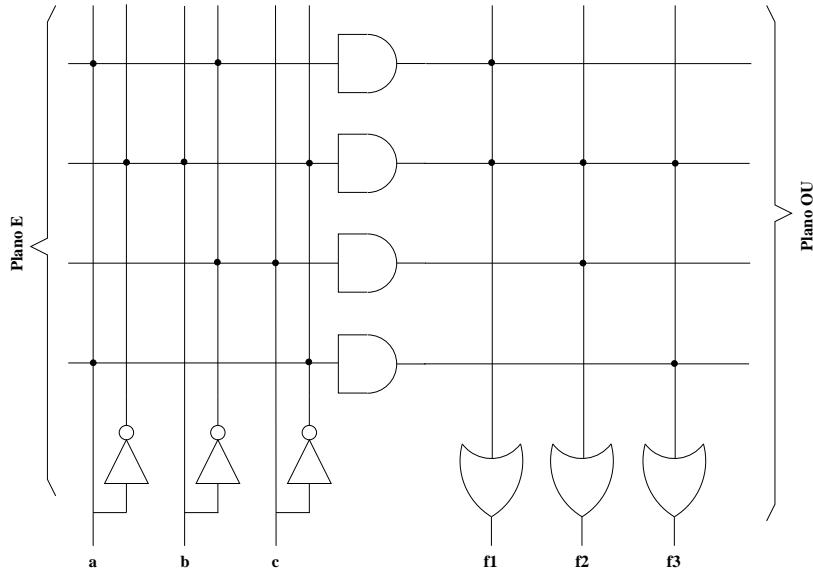
PLAs comerciais têm tipicamente<sup>4</sup> entre 10 e 20 entradas, entre 30 e 60 portas E (produtos) e entre 10 e 20 portas OU (saídas). Em um PLA com 16 entradas, 48 produtos e 8 saídas, existem  $2 \times 16 \times 48 = 1536$  cruzamentos na malha E e  $8 \times 48 = 384$  cruzamentos na malha OU. Um número considerável de funções relativamente complexas podem ser realizadas via um PLA. Claramente, quanto menor o número de variáveis e termos produtos utilizados na expressão de uma função, menor será o “tamanho” do PLA necessário para a realização da função.

Sejam as seguintes funções (já minimizadas individualmente) :

$$f_0 = a$$

$$f_1 = \bar{b}$$

<sup>4</sup>Informação colhida em torno de 2010 ...



**Figura 6.8:** Esquema simplificado de um PLA.

$$f_2 = bc + ab$$

$$f_3 = \bar{a}\bar{b} + \bar{a}c$$

Se expressas desta forma, para implementá-las, serão necessárias 6 portas E. No entanto, note que elas podem ser reescritas como:

$$f_0 = a\bar{b} + ab$$

$$f_1 = \bar{a}\bar{b} + a\bar{b}$$

$$f_2 = \bar{a}bc + ab$$

$$f_3 = \bar{a}\bar{b} + \bar{a}bc$$

e neste caso são necessárias apenas 4 portas E.

Este é outro exemplo no qual mostra-se que o compartilhamento de portas entre múltiplas funções pode levar à redução do número total de portas necessárias para a realização das funções, mesmo que individualmente as expressões não estejam na forma minimal.

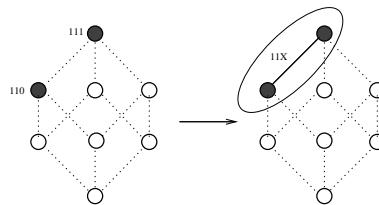
Observe, porém, que no exemplo acima, o número total de produtos foi reduzido às custas de aumento no número total de somas. Se o objetivo é a realização em um PLA (*Programmable logic array*) não é relevante se uma função é formada, por exemplo, por 2 ou 3 termos produto (desde que esse número de produtos não ultrapasse o número de portas E no PLA). Porém, se o objetivo for minimizar também o número de entradas das portas, então é preciso tomar cuidado para não aumentar demasiadamente o número de termos produto (pois isso implicaria

em aumento no número de entradas da porta OU).

## 6.5 Método de Quine-McCluskey (\*)

Mapas de Karnaugh representam uma maneira visual e intuitiva de se minimizar funções booleanas. No entanto, eles só se aplicam a funções com até 6 variáveis e não são sistemáticos (adequados para programação). O algoritmo tabular de Quine-McCluskey para minimização de funções Booleanas é um método clássico que sistematiza este processo de minimização para um número arbitrário de variáveis.

Tanto os mapas de Karnaugh como o algoritmo de Quine-McCluskey (QM) requerem que a função booleana a ser minimizada esteja na forma SOP canônica. A idéia básica do algoritmo QM consiste em encarar os mintermos da SOP canônica como pontos no  $n$ -espaço, ou seja, como vértices de um  $n$ -cubo. A partir do conjunto destes vértices (ou 0-cubos) procura-se gerar todos os 1-cubos possíveis combinando-se dois deles (equivale a gerar as arestas do cubo que ligam dois 0-cubos da função). Na figura 6.9 temos um exemplo no qual os 0-cubos 111 e 110 são agrupados para gerar o 1-cubo 11X. Note, que o 3-cubo (cubo como você conhece) da figura é apenas uma outra representação do mapa de Karnaugh. Este processo de combinar



**Figura 6.9:** Passo elementar do algoritmo de Quine-McCluskey.

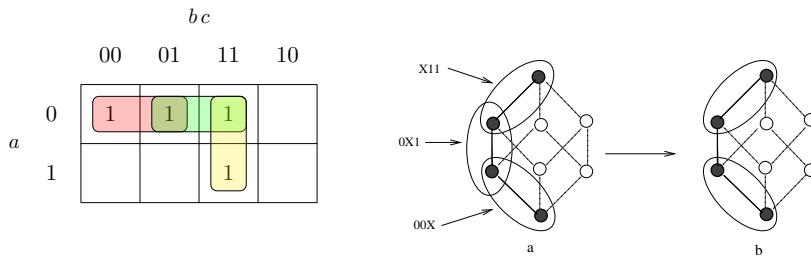
dois cubos corresponde, como já vimos no caso do mapa de Karnaugh, ao processo algébrico de simplificação. Os mintermos da expressão na forma canônica inicial correspondem a o que chamamos de 0-cubo. Combinar dois 0-cubos para gerar um 1-cubo corresponde a combinar dois mintermos para eliminar uma variável e gerar um termo com menos literais para substituí-los. No exemplo da figura 6.9, a simplificação algébrica correspondente é a seguinte:

$$x_1x_2x_3 + x_1x_2\bar{x}_3 = x_1x_2(x_3 + \bar{x}_3) = x_1x_2 \cdot 1 = x_1x_2$$

Enquanto no mapa de Karnaugh pode-se visualmente detectar o maior agrupamento válido de 0-cubos que inclui um 0-cubo específico, no método QM isso é realizado incrementalmente. Isto é, após gerados todos os possíveis 1-cubos, procura-se gerar todos os possíveis 2-cubos por meio da combinação de dois 1-cubos e assim por diante, até que nenhum cubo de dimensão

maior possa ser gerado a partir da combinação de dois cubos de dimensão menor. Note que neste processo estão envolvidos apenas os 0-cubos (vértices do cubo) que correspondem aos mintermos da função (i.e., o valor da função nesses vértices deve ser 1 necessariamente). Os cubos resultantes (aqueles que não foram combinados com nenhum outro) ao final de todo o processo são os **implicantes primos** (ou seja, cubos maximais) da função. Cubos maximais são exatamente os grupos que demarcamos no mapa de Karnaugh (inclusive os redundantes). Veja um exemplo a seguir.

**Exemplo:** Considere a expressão Booleana  $f(a, b, c) = \sum m(0, 1, 3, 7)$ . Os implicantes primos dessa função são  $00X$ ,  $0X1$  e  $X11$ , conforme marcados no mapa de Karnaugh. Graficamente, estes implicantes primos (ou cubos) correspondem respectivamente aos intervalos  $[000, 001]$ ,  $[001, 011]$  e  $[011, 111]$  ilustrados na figura 6.10(a).



**Figura 6.10:** No mapa de Karnaugh, em vermelho temos os cubo  $00X$ , em verde  $0X1$ , e em amarelo  $X11$ . Os implicantes primos são mostrados em (a) e uma cobertura mínima em (b). O cubo  $0X1$  (ou implicante primo  $0X1$  ou ainda intervalo  $[001, 011]$ ) é redundante.

Nem todos os implicantes primos fazem necessariamente parte da forma SOP minimal, como mostrado no exemplo acima. Portanto, o cálculo da forma SOP minimal de uma função booleana no algoritmo QM é dividido em duas etapas:

1. Cálculo de todos os implicantes primos da função
2. Cálculo de uma cobertura (subconjunto dos implicantes primos) mínima da função

O ponto central da segunda etapa é o cálculo de um menor subconjunto do conjunto de implicantes primos, que sejam suficientes para cobrir<sup>5</sup> todos os mintermos da função. Tal conjunto é denominado uma **cobertura mínima**.

No caso de mapas de Karnaugh, estas duas etapas são realizadas conjuntamente de forma um tanto “intuitiva”. No caso do algoritmo QM, estas etapas são realizadas explícita e separadamente. As etapas são descritas a seguir, apoiando-se em exemplos.

<sup>5</sup>Um conjunto de implicantes primos (cubos maximais) cobre um mintermo (0-cubo) se este é coberto por pelo menos um dos implicantes primos.

### 6.5.1 Cálculo de implicantes primos

A primeira etapa do algoritmo QM consiste de um processo para determinação de todos os implicantes primos. A seguir descrevemos os passos que constituem esta etapa, mostrando como exemplo o cálculo dos implicantes primos da função  $f(x_1, x_2, x_3) = \sum m(0, 1, 4, 5, 6)$ .

- Primeiro passo : converter os mintermos para a notação binária.

000, 001, 100, 101, 110

- Segundo passo : Separar os mintermos em grupos de acordo com o número de 1's em sua representação binária e ordená-los em ordem crescente, em uma coluna, separando os grupos com uma linha horizontal.

000
001
100
101
110

- Terceiro passo : combinar todos os elementos de um grupo com todos os elementos do grupo adjacente inferior para geração de cubos de dimensão maior. Para cada 2 grupos comparados entre si, gerar um novo grupo na próxima coluna e colocar os novos cubos. Marcar com  $\checkmark$  os cubos que foram usados para gerar novos cubos.

$\checkmark$	000
$\checkmark$	001
$\checkmark$	100
$\checkmark$	101
$\checkmark$	110

 $\implies$ 

00X
X00
<hr/>
X01
10X
1X0

Observação : o novo cubo gerado será inserido no novo conjunto se e somente se ele ainda não estiver contido nele.

Repetir o processo para cada nova coluna formada, até que nenhuma combinação mais seja possível.

$\checkmark$	000
$\checkmark$	001
$\checkmark$	100
$\checkmark$	101
$\checkmark$	110

 $\implies$ 

$\checkmark$	00X
$\checkmark$	X00
<hr/>	
$\checkmark$	X01
$\checkmark$	10X
<hr/>	
	1X0

 $\implies$ 

X0X
-----

- Quarto passo : Listar os implicantes primos. Os implicantes primos são os cubos que não foram combinados com nenhum outro, ou seja, aqueles que não estão com a marca  $\checkmark$ .

1X0 e X0X

## Cálculo de uma cobertura mínima

Uma cobertura mínima pode ser calculada com a ajuda de uma tabela denominada **Tabela de Implicantes Primos**. Este processo é mostrado a seguir, usando como exemplo a minimização da função  $f(x_1, x_2, x_3, x_4, x_5) = \sum(1, 2, 3, 5, 9, 10, 11, 18, 19, 20, 21, 23, 25, 26, 27)$ .

1. Construir a Tabela de Implicantes Primos: No topo das colunas desta tabela deve-se colocar os mintermos de  $f$  e, à esquerda de cada linha, os implicantes primos. Por simplicidade, no cabeçalho das colunas são colocados os índices decimais dos mintermos (isto é, 3 corresponde a  $m_3$  ou ainda à entrada 00011). Os implicantes primos devem ser listados em ordem decrescente de acordo com a sua dimensão, isto é, em ordem crescente de acordo com o número de literais. Deve-se acrescentar uma coluna à esquerda e uma linha na parte inferior da tabela.

Em cada linha, marcar as colunas com  $\checkmark$  quando o implicante primo da linha cobre o mintermo da coluna.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
	XX01X		$\checkmark$	$\checkmark$			$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$					$\checkmark$	$\checkmark$
	X10X1					$\checkmark$		$\checkmark$							$\checkmark$	$\checkmark$
	0X0X1	$\checkmark$		$\checkmark$		$\checkmark$		$\checkmark$								
	00X01	$\checkmark$			$\checkmark$											
	X0101				$\checkmark$							$\checkmark$				
	1010X										$\checkmark$	$\checkmark$				
	10X11								$\checkmark$				$\checkmark$			
	101X1										$\checkmark$	$\checkmark$				

2. Selecionar os implicantes primos essenciais: deve-se procurar na tabela as colunas que contém apenas uma marca  $\checkmark$ . A linha na qual uma dessas colunas contém a marca  $\checkmark$  corresponde a um implicante primo essencial. Em outras palavras, este implicante primo é o único que cobre o mintermo da coluna e, portanto, não pode ser descartado. Então, deve-se marcar com um asterisco (\*) esta linha na coluna mais à esquerda, para indicar que este é um implicante primo essencial. A seguir, deve-se marcar, na última linha da tabela, todas as colunas cujo mintermo é coberto pelo implicante primo selecionado.

No exemplo, o mintermo 2 é coberto apenas pelo implicante primo  $XX01X$ . Logo  $XX01X$  é essencial.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
*	XX01X		✓	✓			✓	✓	✓	✓					✓	✓
	X10X1					✓		✓						✓		✓
	0X0X1	✓		✓		✓		✓								
	00X01	✓			✓											
	X0101				✓							✓				
	1010X										✓	✓				
	10X11								✓			✓				
	101X1									✓	✓					
			✓	✓			✓	✓	✓	✓					✓	✓

A linha correspondente a um implicante primo essencial, bem como as colunas cujos mintermos são cobertos por esse implicante primo, devem ser desconsiderados no prosseguimento do processo.

Deve-se repetir o processo enquanto existir, na tabela restante, algum implicante primo essencial.

No exemplo, vemos que o mintermo 25 é coberto apenas pelo implicante primo  $X10X1$  e que o mintermo 20 é coberto apenas pelo implicante primo  $1010X$ . Logo, esses dois implicantes primos também são essenciais.

		1	2	3	5	9	10	11	18	19	20	21	23	25	26	27
*	XX01X		✓	✓			✓	✓	✓	✓					✓	✓
*	X10X1					✓		✓						✓		✓
	0X0X1	✓		✓		✓		✓								
	00X01	✓			✓											
	X0101				✓							✓				
*	1010X									✓	✓					
	10X11								✓			✓				
	101X1									✓	✓					
			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

3. Reducir a tabela: eliminar as colunas cujos mintermos já foram cobertos (ou seja, manter apenas as colunas correspondentes aos mintermos não cobertos pelos implicantes primos essenciais). Eliminar as linhas correspondentes aos implicantes primos essenciais e as linhas que não cobrem nenhum dos mintermos restantes na tabela.

No exemplo, após a redução, temos a seguinte tabela:

		1	5	23
	0X0X1	✓		
	00X01	✓	✓	
	X0101		✓	
	10X11			✓
	101X1			✓

4. Selecionar os implicants primos secundariamente essenciais: eliminar as linhas dominadas e as colunas dominantes e, em seguida, selecionar os essenciais.

**Colunas dominantes:** Diz-se que uma coluna  $\beta$  na Tabela de Implicantes Primos domina uma coluna  $\alpha$  se e somente se todos os implicants que cobrem o mintermo da coluna  $\alpha$  cobrem também o mintermo da coluna  $\beta$ . Se  $\beta$  domina  $\alpha$ , então a coluna  $\beta$  pode ser removida da tabela. (Por quê?)

**Linhas dominadas ou equivalentes:** Sejam  $A$  e  $B$  duas linhas na Tabela de Implicantes Primos reduzida. Então dizemos que a linha  $A$  domina  $B$  se o implicant da linha  $A$  cobre, ao menos, todos os mintermos cobertos pelo implicant da linha  $B$ . Dizemos que as linhas  $A$  e  $B$  são equivalentes se os respectivos implicants primos cobrem exatamente os mesmos mintermos na tabela. Se  $A$  domina  $B$ , ou se  $A$  e  $B$  são equivalentes, e **se o custo do implicant da linha  $A$  é menor ou igual ao da linha  $B$**  (ou seja, o implicant da linha  $A$  possui não mais literais que o implicant da linha  $B$ ), então a linha  $B$  pode ser eliminada da tabela (Por quê?).

Após a eliminação de colunas dominantes e linhas dominadas (ou equivalentes), deve-se repetir o mesmo processo do passo 2, porém os implicants primos essenciais serão chamados secundariamente essenciais e marcados com dois asteriscos (\*\*).

No exemplo, a linha do implicant primo  $X0101$  pode ser eliminada pois é dominada pela linha do implicant  $00X01$ . A linha do implicant  $101X1$  pode ser eliminada pois é equivalente a do implicant  $10X11$ . Neste último caso, note que, alternativamente, podemos eliminar a linha do implicant  $10X11$  em vez da linha do implicant  $101X1$ .

		1	5	23
	0X0X1	✓		
**	00X01	✓	✓	
**	10X11			✓
		✓	✓	✓

Deve-se repetir o processo descrito neste passo até que não seja mais possível fazer qualquer eliminação ou até que a tabela fique vazia. Se a tabela não ficar vazia, a tabela restante é chamada **tabela cíclica**.

5. Resolver a tabela cíclica: Para isso, uma possível abordagem é o método de Petrick, um método de busca exaustiva. Ele fornece todas as possíveis combinações dos implicants primos restantes que são suficientes para cobrir os mintermos ainda não cobertos pelos implicants primos já selecionados. Deve-se escolher, dentre essas combinações, uma que envolve o menor número de termos. Caso existam mais de uma nestas condições, deve-se escolher a de custo mínimo (aquela que envolve menor número de literais).

**Exemplo:** Considere a tabela cíclica a seguir:

		0	4	13	15	10	26	16
a	0X10X		✓	✓				
b	011XX			✓	✓			
c	01X1X				✓	✓		
d	1X0X0						✓	✓
e	00X00	✓	✓					
f	X1010					✓	✓	
g	X0000	✓						✓

Para que todos os mintermos da tabela cíclica sejam cobertos, a seguinte expressão deve ser verdadeira.

$$(e + g)(a + e)(a + b)(b + c)(c + f)(d + f)(d + g) = 1$$

Uma soma nesta expressão indica quais implicantes primos poderiam ser escolhidos para cobrir um mintermo (uma coluna). O produto de somas expressa, portanto, todas as combinações possíveis de escolha para se cobrir todas as colunas. Transformando esta expressão em soma de produtos, obtemos todas as possíveis soluções (cada produto é uma solução viável). Dentre os produtos, deve-se escolher aquele(s) de menor custo (menor número de implicantes primos e implicantes com menor número de literais). (Se eu não errei nos cálculos, as soluções são  $\{a, c, d, e\}$ ,  $\{b, c, d, e\}$  e  $\{a, c, d, g\}$  pois os outros tem custo maior).

**Outro exemplo:** Considere a tabela cíclica a seguir e suponha que o custo de  $A$  é menor que o de  $B$  e que o custo de  $C$  é menor que o de  $D$ .

	$m_1$	$m_2$	$m_3$
A	✓		
B	✓	✓	
C			✓
D		✓	✓

Então as possíveis soluções são  $(A+B)(B+D)(C+D) = (AB+AD+B+BD)(C+D) = (B+AD)(C+D) = BC+BD+ACD+AD$ . Dos que envolvem dois implicantes, certamente o custos de  $BC$  e de  $AD$  são menores que o custo de  $BD$ . Então a escolha final fica entre  $BC$  e  $AD$ .

#### Resumo do procedimento para cálculo de cobertura mínima:

1. Montar a tabela de implicantes primos
2. Identificar todos os implicantes primos essenciais e eliminar as linhas correspondentes, bem como as colunas dos mintermos cobertos por esses implicantes.

3. Eliminar colunas dominantes: Se uma coluna  $\beta$  tem  $\checkmark$  em todas as linhas que uma outra coluna  $\alpha$  tem  $\checkmark$ , a coluna  $\beta$  é dominante e pode ser eliminada (pois se escolhermos um implicante primo que cobre  $\alpha$ ,  $\beta$  será necessariamente coberto também).
4. Eliminar linhas dominadas ou equivalentes: se uma linha  $A$  tem  $\checkmark$  em todas as colunas em que a linha  $B$  tem  $\checkmark$ , então a linha  $A$  domina a linha  $B$ . Se elas tem  $\checkmark$  exatamente nas mesmas colunas, então elas são equivalentes. Se, além disso, o número de literais de  $A$  é menor que o de  $B$ , então a linha  $B$  pode ser eliminada (pois se tivéssemos uma cobertura envolvendo  $B$ , ao trocarmos  $B$  por  $A$  na cobertura teríamos uma cobertura de menor custo).

Observação: Se o objetivo da minimização é encontrar apenas UMA solução minimal (e NÃO TODAS), então podemos eliminar uma linha  $B$  se existe uma linha  $A$  tal que  $A$  domina  $B$ , ou  $A$  é equivalente a  $B$ , e ambos têm um mesmo custo.

5. Identificar os implicantes essenciais secundários e eliminar as linhas correspondentes, bem como as colunas dos mintermos cobertos por esses implicantes.
6. Repetir 3, 4 e 5 enquanto possível
7. Se a tabela não estiver vazia, aplicar o método de Petrick (que lista todas as possíveis soluções para o restante da tabela) e escolher uma solução de custo mínimo.

**Exemplo:** Considere a função  $f(a, b, c) = a\bar{b}c + \bar{a}b\bar{c} + ab\bar{c} + a\bar{b}c = \sum m(2, 5, 6, 7)$ .

Podemos realizar a simplificação algébrica da seguinte forma:

$$\begin{aligned}
 f(a, b, c) &= a\bar{b}c + \bar{a}b\bar{c} + ab\bar{c} + a\bar{b}c \\
 &= a\bar{b}c + abc + \bar{a}b\bar{c} + ab\bar{c} + a\bar{b}c + abc \\
 &= ac + b\bar{c} + ab \\
 &= ac + b\bar{c}
 \end{aligned}$$

Por QM temos

$\checkmark$	010
$\checkmark$	101
$\checkmark$	110
$\checkmark$	111

 $\Rightarrow$ 

X10
1X1
11X

Os implicantes primos são  $X10$  ( $b\bar{c}$ ),  $1X1$  ( $ac$ ) e  $11X$  ( $ab$ ). Uma cobertura mínima pode ser calculada usando-se a Tabela de Implicantes Primos.

		2	5	6	7
*	X10	✓		✓	
*	1X1		✓		✓
	11X			✓	✓
		✓	✓	✓	✓

Os implicants primos  $X10$  e  $1X1$  são essencias e cobrem todos os mintermos da função. Logo formam uma cobertura mínima.

### 6.5.2 Funções incompletamente especificadas

Para minimizar uma função incompletamente especificada pelo algoritmo QM, é interessante considerarmos todos os *don't cares* na etapa de cálculo dos implicants primos, pois isto aumenta a chance de se obter cubos maiores (portanto produtos com menos literais). Observe que, durante as iterações para a geração dos implicants primos, um cubo que cobre apenas *don't cares* não deve ser eliminado pois ele pode, eventualmente em iterações futuras, se juntar a outro cubo para formar outro cubo maior.

De forma mais genérica do que a vista anteriormente, podemos caracterizar uma função booleana  $f$  através dos seus conjuntos um, zero e dc (de *don't care*), definidos respectivamente por  $f\langle 1 \rangle = \{\mathbf{x} \in \{0, 1\}^n : f(\mathbf{x}) = 1\}$ ,  $f\langle 0 \rangle = \{\mathbf{x} \in \{0, 1\}^n : f(\mathbf{x}) = 0\}$  e  $f\langle * \rangle = \{0, 1\}^n \setminus (f\langle 1 \rangle \cup f\langle 0 \rangle)$ .

Na parte de cálculo dos implicants primos devem ser utilizados todos os elementos de  $f\langle 1 \rangle \cup f\langle *\rangle$ . Na parte de cálculo de uma cobertura mínima devem ser considerados apenas os elementos de  $f\langle 1 \rangle$ .

## 6.6 Outros algoritmos de minimização lógica 2-níveis

Algoritmos de minimização tabular (como o Quine-McCluskey) têm as seguintes características:

- eles listam explicitamente todos os implicants primos, cuja quantidade pode ser de ordem exponencial no número de variáveis  $n$
- requerem que a função a ser minimizada esteja na forma SOP canônica. Não é raro que, juntamente com os *don't cares*, o número de produtos canônicos seja da ordem de  $2^{n-1}$
- a tabela-cíclica pode ser bem grande (e não há algoritmo eficiente para o cálculo da solução ótima)

Do ponto de vista computacional, isto implica em consumo de grande quantidade de memória e tempo de processamento longo.

Muito esforço ocorreu nas décadas de 1980 e início da década de 1990 no sentido de se desenvolver programas para minimização de funções com várias variáveis, motivados pelos PLAs. Os esforços realizados foram no sentido de achar alternativas que não requeressem o cálculo explícito de todos os implicantes primos, formas eficientes para o cálculo de coberturas mínimas e heurísticas eficientes nesses processos.

Um dos resultados do esforço foi a criação do algoritmo ESPRESSO (desenvolvido por um grupo da University of California – Berkeley [Brayton et al., 1984, McGreer et al., 1993]) que foi referência para minimização lógica dois-níveis durante um tempo. Para maiores detalhes veja [Brayton et al., 1984], capítulo 7 de [Micheli, 1994], seção 6.10 de [Hill and Peterson, 1993]. ESPRESSO não requer que a função a ser minimizada esteja na forma SOP canônica, não calcula os implicantes primos explicitamente, e utiliza uma série de heurísticas. Além disso, ele também realiza a minimização conjunta de múltiplas funções e de funções multi-valoradas (lógica multi-valores). Depois do ESPRESSO, foram propostas outras melhorias (como o uso de *BDD — Binary Decision Diagrams*) por Coudert e outros [Coudert, 1994, Coudert, 1995], e o BOOM [Hlavicka and Fiser, 2001, Fišer and Hlavicka, 2003]<sup>6</sup>

---

<sup>6</sup>Não tenho conhecimentos sobre desenvolvimentos posteriores ao BOOM.

# Capítulo 7

## Circuitos combinacionais

Última atualização em 09/05/2018

O termo **combinacional** no título é utilizado na área de circuitos lógicos em contraste aos circuitos do tipo **sequencial**. Nos circuitos combinacionais, os valores de saída são determinados unicamente apenas a partir dos valores de entrada. Todos os circuitos que vimos até este momento são do tipo combinacional. Já os circuitos sequenciais são aqueles capazes de “armazenar” um valor (manter um certo estado) e nos quais os valores de saída são determinados em função não apenas dos valores de entrada, mas também do estado atual do circuito. Circuitos sequenciais serão estudados em um capítulo mais adiante.

Neste capítulo serão apresentados alguns circuitos combinacionais básicos, úteis como componentes “caixa-preta” na construção de circuitos complexos.

### 7.1 Somador

O circuito somador já foi discutido no capítulo 3. Ele é parte fundamental de qualquer ULA (unidade lógico-aritmética).

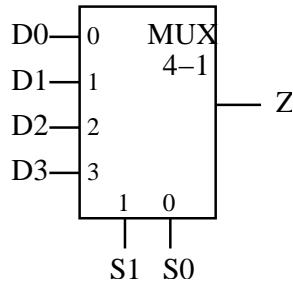
### 7.2 Comparador

Outro circuito que geralmente está presente em uma ULA é o circuito comparador. A construção de um circuito comparador para números de 8 *bits* na representação complemento de 2 é o tema do EP2.

## 7.3 Multiplexadores e demultiplexadores

### 7.3.1 Multiplexadores

**Multiplexadores** (também conhecidos como seletores de dados ou MUX) são circuitos combinatoriais com  $n$  entradas de dados, uma saída e entradas de controle denominados seletores. A saída desse circuito é sempre uma cópia de uma das entradas. Dentre as  $n$  entradas, a que é enviada para a saída é justamente aquela cujo índice é igual ao valor codificado nos *bits* seletores. A figura 7.1 mostra um multiplexador  $4 \times 1$ , isto é, um multiplexador de 4 entradas.



**Figura 7.1:** Multiplexador de 4 entradas.

O dado a ser enviado para a saída depende do valor dos seletores e está especificado na tabela a seguir:

$s_1\ s_0$	$z$
00	$D_0$
01	$D_1$
10	$D_2$
11	$D_3$

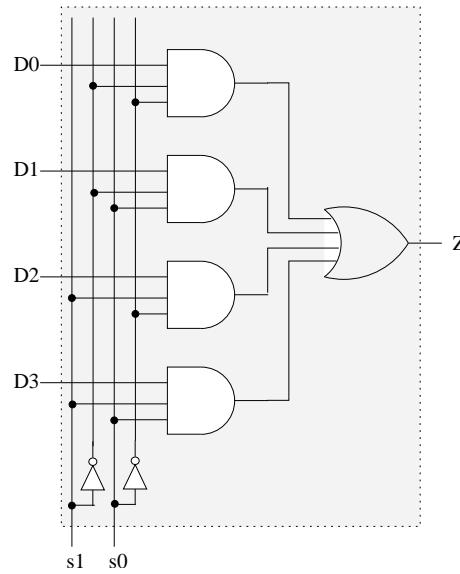
Podemos observar que  $z$  é uma função dos *bits* seletores  $s_1, s_0$  e dos dados de entrada  $D_0, D_1, D_2, D_3$  (que podem ser pensadas como variáveis lógicas). Assim, podemos escrever  $z$  como

$$z(D_0, D_1, D_2, D_3, s_1, s_0) = D_0 \bar{s}_1 \bar{s}_0 + D_1 \bar{s}_1 s_0 + D_2 s_1 \bar{s}_0 + D_3 s_1 s_0$$

Note que para cada possível combinação de valores de  $s_1\ s_0$ , apenas um dos produtos toma valor 1 na equação acima. Por exemplo, se  $s_1\ s_0 = 10$ , apenas o mintermo  $s_1 \bar{s}_0$  terá valor 1 e teremos então  $z = D_2$ . O mesmo raciocínio se aplica para os demais valores de  $s_1\ s_0$ .

Uma realização simples de multiplexadores é na forma de circuitos dois-níveis, consistindo de  $n$  portas E no primeiro nível e uma porta OU no segundo nível. A realização do multiplexador de 4 entradas é mostrado na figura 7.2.

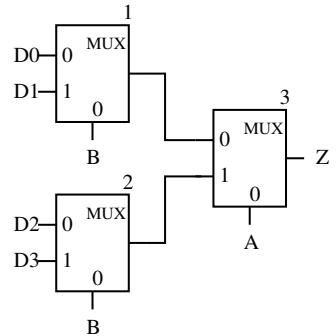
De forma geral, se um multiplexador possui  $k$  *bits* seletores, pode possuir até  $2^k$  entradas (ou,



**Figura 7.2:** Circuito de um multiplexador de 4 entradas.

observado de outro ponto de vista, se possui  $n$  entradas de dados, deve possuir  $k \geq \log_2 n$  bits seletores). No exemplo acima, de 4 entradas, 2 bits seletores são necessários e suficientes.

Um MUX pode ser realizado como composição de múltiplos MUX com um menor número de entradas cada. Por exemplo, um MUX  $4 \times 1$  pode ser realizado usando três MUX  $2 \times 1$ , conforme ilustrado na figura 7.3.



**Figura 7.3:** Realização de um MUX  $4 \times 1$  como composição de três MUX  $2 \times 1$ .

Note que se  $AB = 00$ , a saída do MUX 1 é  $D_0$  e a do MUX 2 é  $D_2$  (já que  $B = 0$ ) e a saída do MUX 3 é  $D_0$  (já que  $A = 0$  também). Se  $AB = 01$  então as saídas são, respectivamente,  $D_1$ ,  $D_3$  e  $D_1$ . Se  $AB = 10$ , então as saídas são, respectivamente,  $D_0$ ,  $D_2$  e  $D_2$ . Finalmente, se  $AB = 11$ , então as saídas são, respectivamente,  $D_1$ ,  $D_3$  e  $D_3$ . Em todos os casos, a saída  $z$  da composição (que é a saída do MUX 3) é a mesma de um MUX  $4 \times 1$  quando o par  $AB$  é usado como entrada para os seus bits seletores.

**Pergunta:** E se trocarmos a variável de entrada dos seletores na figura acima? Se colocarmos

$A$  no seletor dos MUX 1 e MUX 2 e  $B$  na do MUX 3, ainda é possível realizarmos um MUX  $4 \times 1$  no qual os bits seletores são alimentados pelo par  $AB$ ?

### 7.3.2 Demultiplexadores

**Demultiplexadores** (também conhecidos como distribuidores de dados ou DMUX) são circuitos combinacionais inversos aos multiplexadores, isto é, possuem apenas uma entrada que é transmitida a apenas uma das  $n$  saídas, determinada pelos *bits* seletores. De forma similar ao MUX, a saída para a qual a entrada é enviada é aquela cujo índice corresponde ao valor definido nos *bits* seletores. Se o demultiplexador possui  $n$  saídas, então são necessários  $k$  seletores, com  $2^k \geq n$ .

Uma implementação simples de DMUX consiste de  $n$  portas E. Cada porta E gera uma saída  $z_i$ ,  $i = 0, 1, 2, \dots, n$ , que corresponde à função dada por

$$z_i(D, s_{k-1}, \dots, s_1, s_o) = D m_i$$

na qual  $D$  é a variável de entrada e  $m_i$  é o mintermo correspondente a  $i$ . Por exemplo, se  $k = 2$  e  $i = 1$ , temos  $z_1 = D \bar{s}_1 s_o$  ( $m_1 = \bar{s}_1 s_o$ ).

### 7.3.3 Exemplos de utilização de MUX e DMUX

**ULA** Adição e subtração são operações realizadas por qualquer ULA. Quando consideramos a notação complemento de dois, vimos que o mesmo circuito somador pode ser usado tanto para realizar a operação de adição ( $A + B$ ) quanto a de subtração ( $A - B$ ). Caso a operação a ser executada seja a de subtração, então  $B$  deve ser complementado antes de alimentar o circuito. Ou seja, existe a necessidade de se chavear o tráfego de sinais no circuito de forma que ora  $B$  ou ora  $\bar{B}$  seja enviado ao circuito somador, de acordo com o cálculo a ser efetuado. Este chaveamento pode ser realizado por meio de um MUX, que recebe nas entradas  $B$  e  $\bar{B}$  e envia para a saída uma delas, e tendo como entrada no *bit* seletor um *bit* que indica a operação a ser executada.

**Transmissão serial** Suponha que temos dois subsistemas  $S$  e  $R$ , sendo que  $S$  possui  $n$  geradores cujos sinais gerados devem ser transmitidos aos respectivos  $n$  receptores em  $R$ . A transmissão dos  $n$  sinais poderia ser realizada par a par simultaneamente caso existisse um canal de comunicação direta entre os geradores de  $S$  e os respectivos receptores em  $R$ . No entanto, os sistemas  $S$  e  $R$  podem estar conectados por apenas um canal de transmissão, o que impossibilitaria a transmissão par a par simultânea. Nestes casos, uma possível solução é uma transmissão serial,

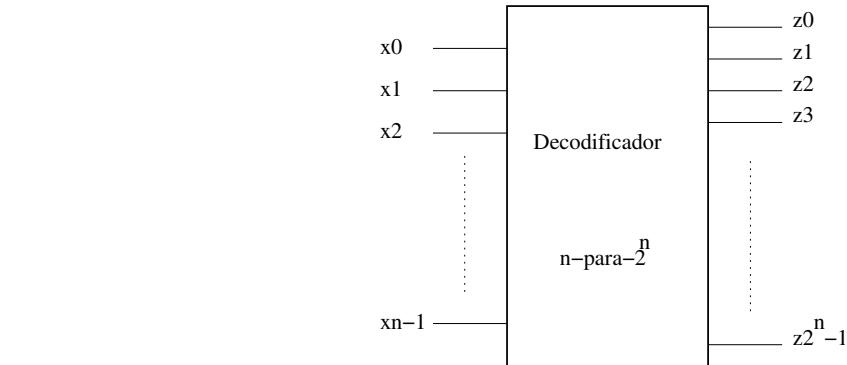
entre um par por vez, e ciclando-se sobre todos os pares. Para tanto, pode-se utilizar um multiplexador acoplado a  $S$  antes do canal de transmissão e, na outra ponta, um demultiplexador acoplado entre o canal de transmissão e o sistema  $R$ . Nesta solução, os seletores devem estar sincronizados, isto é, os seletores de ambos os sistemas devem estar com o mesmo valor. Se o valor dos seletores do MUX e do DMUX for  $i$ , o MUX selecionará o sinal do gerador  $i$  para ser transmitido, e na outra ponta, o DMUX irá direcionar o sinal recebido para o receptor  $i$ . Alterando-se o valor de  $i$  ciclicamente, tem-se a transmissão serial.

## 7.4 Decodificadores e codificadores

### 7.4.1 Decodificadores

**Decodificadores** são circuitos combinacionais com  $n$  entradas e  $2^n$  saídas. As  $n$  entradas são interpretadas como um número em binário e portanto elas podem codificar os números de 0 a  $2^n - 1$ . Apenas uma única saída é ativada; justamente aquela cujo índice corresponde ao valor codificado na entrada. A figura 7.4 mostra um esquema genérico de um decodificador  $n : 2^n$ .

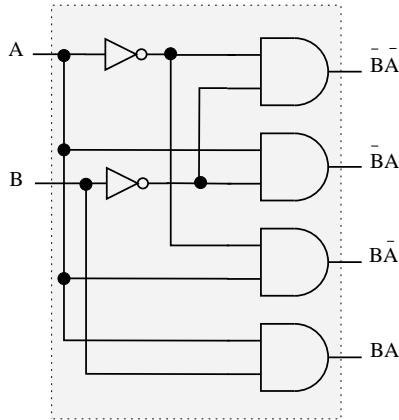
As entradas  $x_{n-1} \dots x_1 x_0$  são interpretadas como um número binário entre 0 e  $2^n - 1$  e tem-se

$$z_i = 1 \iff \sum_{k=0}^{n-1} x_k 2^k = i.$$


**Figura 7.4:** Esquema de um decodificador  $n : 2^n$ .

**Exemplo:** Seja o decodificador  $2 : 4$  mostrado na figura 7.5. Neste caso, o bit menos significativo na entrada recebe  $A$  e o mais significativo recebe  $B$ . Temos  $z_0 = \overline{B} \overline{A}$ ,  $z_1 = \overline{B} A$ ,  $z_2 = B \overline{A}$  e  $z_3 = BA$ . Para valores fixos na entrada, apenas um desses produtos canônicos tomará valor 1 (ou seja, apenas uma saída será ativada).

Conceitualmente, o circuito acima poderia ser estendido para realizar decodificadores  $n : 2^n$ , para um valor de  $n$  arbitrariamente grande. No entanto, na prática existem limitações tecnológicas

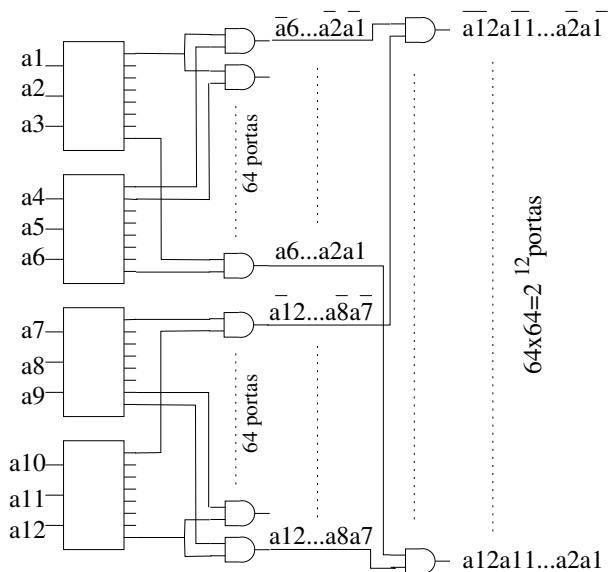


**Figura 7.5:** Circuito de um decodificador  $2 : 4$ .

(físicas) conhecidas como *fan-in* (número máximo de entradas possíveis em uma porta lógica) que restringem este valor  $n$ . Para valores grandes de  $n$ , decodificadores são realizados por circuitos multi-níveis, conforme veremos a seguir.

#### 7.4.2 Realização multi-níveis de decodificadores\*

Vimos que decodificadores possuem  $n$  entradas e  $2^n$  saídas e que sua realização trivial utiliza  $2^n$  portas E, com  $n$  entradas cada. Para contornar o problema de *fan-in* (número máximo de entradas possíveis em uma porta lógica), decodificadores com grande número de entradas podem ser realizados por circuitos com múltiplos níveis. A figura 7.6 mostra como pode ser realizado um decodificador  $12 : 2^{12}$  em três níveis.



**Figura 7.6:** Realização três-níveis de um decodificador  $12 : 2^{12}$ .

No primeiro nível são usados 4 decodificadores 3 : 8. No segundo nível, 64 portas E de duas entradas são usadas para combinar cada uma das 8 saídas do primeiro decodificador com cada uma das 8 saídas do segundo decodificador. A mesma coisa para as saídas do terceiro com as do quarto decodificador. Cada uma das saídas das primeiras 64 portas E do segundo nível são combinadas com cada uma das saídas das últimas 64 portas E no mesmo nível, resultando em um total de  $64 \times 64 = 2^{12}$  portas E no terceiro nível. As saídas dessas  $2^{12}$  portas E correspondem aos produtos canônicos de 12 variáveis.

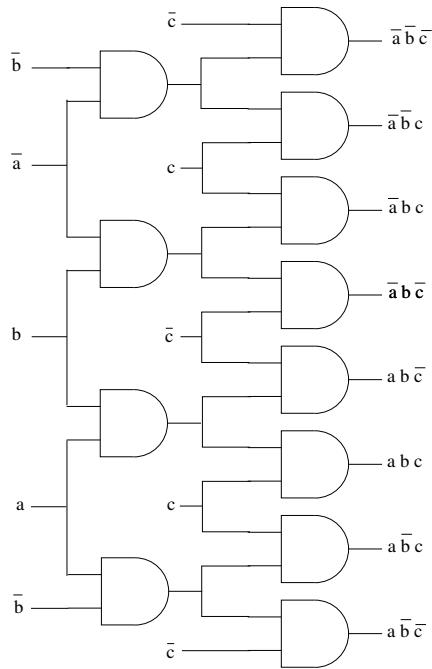
A solução acima utiliza portas E com três entradas no primeiro nível e portas E com duas entradas nos demais níveis. Se o circuito fosse realizado em apenas um nível, as portas E teriam 12 entradas.

Em uma outra possível realização, poderíamos substituir as 128 portas E de duas entradas no segundo nível acima por  $2^{12}$  portas E de quatro entradas e eliminar as portas do terceiro nível. Isto aparentemente reduziria o número total de portas, mas uma vez que  $2^{12}$  domina de longe 128 e uma vez que as portas agora teriam quatro entradas em vez de duas, não se pode dizer que há economia no custo total.

Um outro problema devido às limitações tecnológicas é o conhecido por *fan-out* (número máximo de portas que podem ser alimentadas por uma saída de uma porta lógica). No caso da realização três-níveis do decodificador 12 :  $2^{12}$  visto acima, as saídas das portas no segundo nível alimentam 64 portas no terceiro nível.

Para contornar o *fan-out*, uma possível solução são as realizações em estruturas de árvore. A figura 7.7 mostra a realização de um decodificador 3 : 8 em uma estrutura de árvore. Em vez de termos todas as variáveis alimentando portas no primeiro nível, temos variáveis que alimentam portas nos outros níveis. Usando este esquema, pode-se reduzir o número de portas no próximo nível que precisam ser alimentadas pela saída de uma porta no nível anterior. Mesmo assim, o problema de *fan-out* não é totalmente eliminado pois as variáveis introduzidas nos níveis posteriores do circuito precisam alimentar muitas portas. No entanto, é mais fácil controlar o sinal de algumas poucas entradas (variáveis) para que eles sejam capazes de alimentar um maior número de portas do que fazer o mesmo com as saídas das portas lógicas. Esta solução possui um maior número de níveis e um maior número de portas lógicas do que o esquema mostrado na figura 7.6, mas para decodificadores de muitas entradas pode ser a única solução.

Na prática, as realizações de decodificadores para um número grande de entradas é baseada em uma combinação das estruturas da figura 7.6 e da figura 7.7.



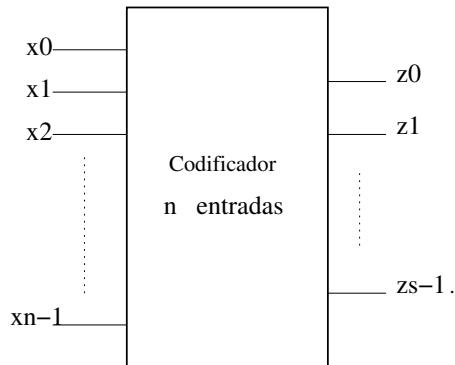
**Figura 7.7:** Decodificador em estrutura de árvore.

### 7.4.3 Codificadores

**Codificadores** são circuitos combinacionais que são o inverso de decodificadores. Um codificador de  $n$  entradas deve possuir  $s$  saídas satisfazendo

$$2^s \geq n \quad \text{ou} \quad s \geq \log_2 n$$

Usualmente deve-se ter apenas uma entrada ativa e a saída será o código binário correspondente à entrada. Isto é, se a  $i$ -ésima entrada estiver ativa, a saída será o código binário de  $i$ . A figura 7.8 mostra o esquema de um codificador de  $n$  entradas. Codificadores com  $s$  saídas



**Figura 7.8:** Esquema de um codificador.

podem ser realizados com portas OU, uma para cada saída.

Embora usualmente os codificadores sejam definidos como um circuito no qual apenas uma entrada encontra-se ativa, é possível termos codificadores com propósitos específicos que, por exemplo, para certos tipos de combinação de entradas gera um dado código de saída e para outras combinações de entradas gera outro código de saída.

#### 7.4.4 Exemplos de utilização de codificadores e decodificadores

**Memórias ROM:** Um exemplo de uso de decodificadores são as memórias do tipo ROM (*Read-Only Memory*). Um diagrama que ilustra uma possível realização de uma ROM com 4 posições e cada posição com capacidade para armazenar 4 bits é mostrada na figura 7.9. As posições da memória podem ser identificadas por “endereços” numéricos; no caso, 0, 1, 2, e 3. Por exemplo, a posição 0 está destacada no diagrama, e nessa posição está armazenado o valor 1110 (na prática, quando há uma bolinha preta, significa que as linhas horizontal e vertical estão conectadas e portanto haverá condução de sinal elétrico). O endereço dessas 4 posições pode ser codificado em dois bits  $x_1 x_0$ . Por exemplo, se  $x_1 x_0 = 00$ , o decodificador irá ativar a saída 0 (a primeira de cima para baixo), o que fará com que o sinal trafegue na linha horizontal e em direção a  $z_3, z_2, z_1$  e  $z_0$  (mas não  $z_0$ ). Teremos, neste caso,  $z_3 z_2 z_1 z_0 = 1110$  que é exatamente o “valor” armazenado na posição 0 da ROM.

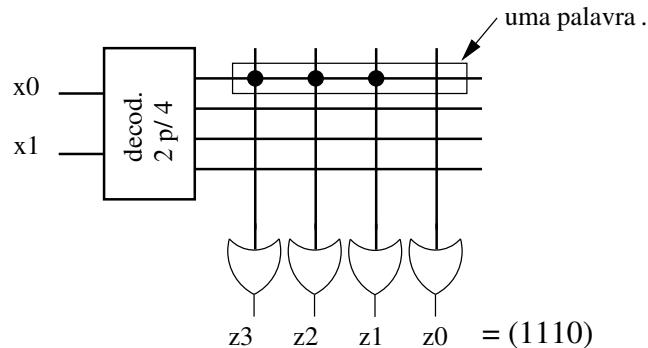


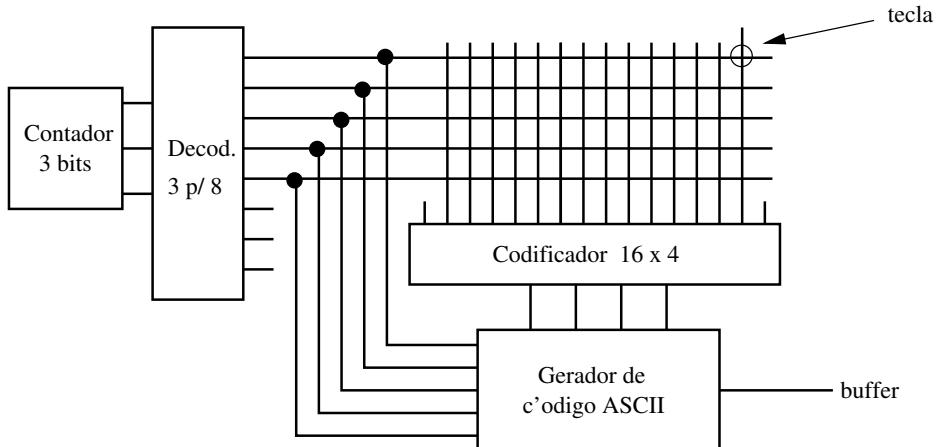
Figura 7.9: Esquema de uma memória ROM com 4 posições e palavras de 4 bits.

De forma geral, a cada endereço ( $x_1 x_0$ ) fornecido como entrada do decodificador, apenas uma saída do decodificador ficará ativa. A palavra na posição de memória endereçada (isto é, o dado armazenado na linha de saída ativada) será transmitida para a saída ( $z_3 z_2 z_1 z_0$ ) (note que o esquema da figura está simplificado; a rigor, cada porta OU possui exatamente 4 entradas que poderão ou não estar conectadas a cada uma das linhas de saída do decodificador).

A estrutura das memórias ROM é semelhante aos PLAs (i.e., um plano de portas E e um plano de portas OU). Na ROM, as portas E estão no decodificador, e há necessariamente  $2^n$  portas E (todos os produtos são canônicos), enquanto num PLA o número de portas E pode variar e elas são programáveis. Na ROM apenas o plano OU pode ser programável. Se o plano OU tem

conexões fixas, trata-se de uma ROM. Se o plano OU puder ser programado, então trata-se de uma PROM (*Programmable ROM*), e se o plano OU puder ser reprogramado trata-se de uma EPROM (*Erasable Programmable ROM*).

**Teclado:** Decodificadores e codificadores podem também ser usados, por exemplo, em teclados. Suponha por exemplo que um teclado simplificado possui 70 teclas. O computador deve ser capaz de saber qual tecla foi pressionada pelo usuário, para gerar o código (por exemplo ASCII) do caractere correspondente à tecla pressionada para ser enviado para o processador. Ao invés de usarmos 70 linhas conectando cada uma das teclas a um gerador de código ASCII, podemos ter um esquema como o ilustrado na figura 7.10, que aproveita o arranjo “bidimensional” (linhas × colunas) das teclas.



**Figura 7.10:** Esquema de um teclado. O decodificador identifica a linha e o codificador a coluna da tecla pressionada. É esperado que a cada tecla pressionada, o contador na entrada do decodificador execute um ciclo e, portanto, ative cada uma das 8 saídas do decodificador uma única vez.

Vamos examinar os componentes desse esquema. Ao lado esquerdo está um contador de 3 *bits*. Estudaremos contadores mais adiante e aqui nos limitaremos apenas a descrever seu funcionamento. O contador no diagrama simplesmente gera sucessivamente os números de 0 a 7 e de forma cíclica (ao atingir 7, volta para 0 e continua). Esses números gerados pelo contador alimentam o decodificador. Isto significa que as saídas do decodificador serão ativadas sucessivamente de 0 a 7, uma por vez, e de forma cíclica também. Vamos supor também que o contador completa um ciclo (e apenas um ciclo) durante o período de tempo no qual uma tecla está pressionada. No diagrama, é mostrado em destaque a tecla no canto superior direito. Quando essa tecla estiver pressionada e a saída 0 do decodificador for ativada, o sinal será transmitido pela linha vertical até o codificador. O codificador é usado para detectar em qual das colunas encontra-se a tecla pressionada, gerando o código binário correspondente ao número da coluna (como temos 14 colunas, 4 *bits* são suficientes na saída do codificador). O gerador de código ASCII recebe a indicação de qual é a linha (via entrada lateral) e a coluna (via saída do co-

dificador) da tecla pressionada. Com essas informações, ele sabe qual tecla foi pressionada e poderá então gerar o código correspondente à tecla. Um teclado de verdade é mais complexo do que isto, especialmente com respeito a aspectos relacionados com sincronização (em situações de pressão continuada de uma tecla, como deve ser definido quantas vezes a tecla foi pressionada?, como lidar com combinações tais como CTRL+ALT?, etc).

## 7.5 Realização de funções arbitrárias

Da mesma forma que funções lógicas representadas por expressões na forma soma de produtos ou produto de somas podem ser facilmente implementados em PLAs, circuitos disponíveis como MUX ou decodificadores podem ser usados para a realização de funções quaisquer.

### 7.5.1 Realização de funções com MUX

Para realizar funções por meio de MUX, devemos escolher as variáveis que funcionarão como seletores. Baseado nisso, determina-se quais subexpressões devem ser enviadas às entradas de dados do MUX, para que tenhamos a realização da função.

**Exemplo:** Suponha que desejamos realizar a função  $f(a, b, c) = \bar{a}\bar{b} + a c$  usando um MUX  $4 \times 1$  e que as variáveis  $a$  e  $b$  serão utilizadas nos seletores. Para determinar quais subexpressões devem ser enviadas às entradas de dados do MUX, podemos expandir a expressão da função de forma que os literais correspondentes às variáveis  $a$  e  $b$  apareçam em todos os produtos da expressão resultante. No caso da função dada, temos:

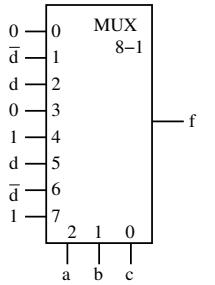
$$\begin{aligned} f(a, b, c) &= \bar{a}\bar{b} + a c \\ &= \bar{a}\bar{b} + a(b + \bar{b})c \\ &= \bar{a}\bar{b} + abc + a\bar{b}c \\ &= (\bar{a}\bar{b})1 + (\bar{a}b)0 + (a\bar{b})c + (ab)c \end{aligned}$$

Na última linha os produtos envolvendo as variáveis  $a$  e  $b$  foram destacadas para facilitar o entendimento. A realização em um MUX  $4 \times 1$  pode ser derivada diretamente dessa última expressão: fazendo-se  $s_1 = a$ ,  $s_0 = b$ , basta fazermos  $D_0 = 1$ ,  $D_1 = 0$ ,  $D_2 = c$  e  $D_3 = c$ .

### 7.5.2 Realização multi-níveis de funções com MUX

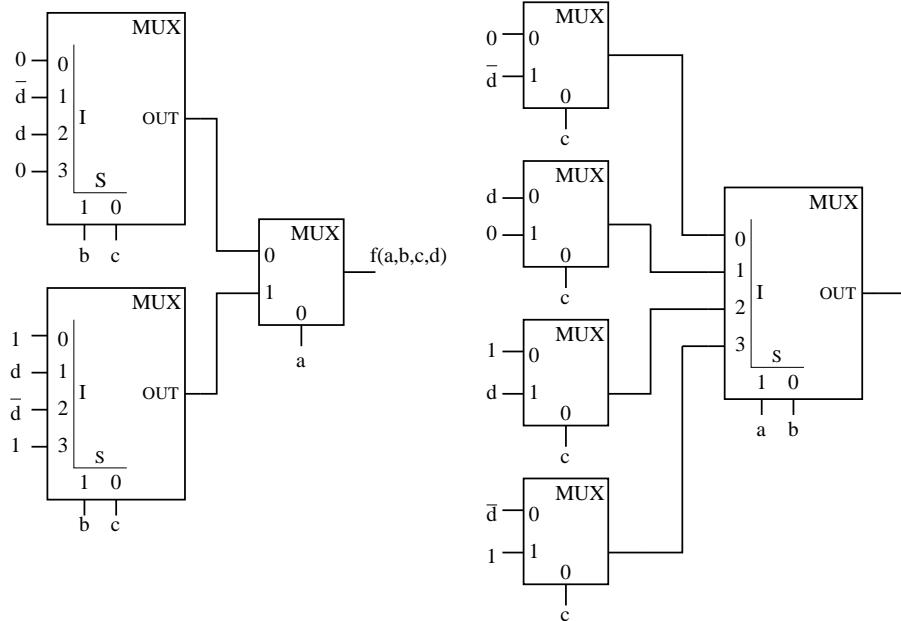
Considere por exemplo a função  $f(a, b, c, d) = \sum m(2, 5, 8, 9, 11, 12, 14, 15)$ . Se procedermos como feito acima, mas desta vez usando um MUX  $8 \times 1$ , uma possível realização (usando  $a b c$

nos seletores) é mostrada na figura 7.11.



**Figura 7.11:** Uma realização de  $f(a, b, c, d) = \sum m(2, 5, 8, 9, 11, 12, 14, 15)$  com um MUX  $8 \times 1$ .

Porém, podemos também utilizar MUX arranjados em múltiplos níveis. Por exemplo, para a mesma função  $f$  acima, duas possíveis realizações usando MUX  $4 \times 1$  e MUX  $2 \times 1$  são mostradas na figura 7.12.



**Figura 7.12:** Realização de  $f(a, b, c, d) = \sum m(2, 5, 8, 9, 11, 12, 14, 15)$  com (a) dois MUX  $4 \times 1$  no primeiro nível e um MUX  $2 \times 1$  no segundo nível e (b) quatro MUX  $2 \times 1$  no primeiro nível e um MUX  $4 \times 1$  no segundo nível.

Estas estruturas podem ser obtidas a partir da análise dos mintermos arranjados em forma tabular, conforme mostrado a seguir. A tabela da esquerda considera o uso das entradas  $b$  e  $c$  como seletores dos MUX  $4 \times 1$  no primeiro nível e o uso da variável  $a$  como seletor do MUX  $2 \times 1$  do segundo nível (diagrama à esquerda na figura 7.12). A tabela da direita faz o análogo para a implementação com quatro MUX  $2 \times 1$  no primeiro nível e um MUX  $4 \times 1$  no segundo nível (diagrama à direita na figura 7.12).

$abcd$	$a$	$bc$	$d$	input
0010	0	01	0	$\bar{d}$
0101	0	10	1	$d$
1000	1	00	0	$\bar{d} + d = 1$
1001	1	00	1	
1011	1	01	1	$d$
1100	1	10	0	$\bar{d}$
1110	1	11	0	$\bar{d} + d = 1$
1111	1	11	1	

$abcd$	$ab$	$c$	$d$	input
0010	00	1	0	$\bar{d}$
0101	01	0	1	$d$
1000	10	0	0	$\bar{d} + d = 1$
1001	10	0	1	
1011	10	1	1	$d$
1100	11	0	0	$\bar{d}$
1110	11	1	0	$\bar{d} + d = 1$
1111	11	1	1	

Outra abordagem para determinar a estrutura hierárquica dos MUXes na realização de funções com múltiplos níveis de MUXes é baseada na aplicação sucessiva da expansão de Shannon. Um exemplo simples dessa abordagem foi mostrado na seção 7.5.1. Aqui mostramos que dependendo da sequência de variáveis em torno das quais a expansão é aplicada, pode-se chegar a diferentes estruturas multi-níveis.

Conforme já vimos, o teorema de **Expansão de Shannon** afirma que qualquer função  $f$  de  $n$  variáveis pode ser escrita em termos de funções de  $n - 1$  variáveis da seguinte forma:

$$f(x_1, \dots, x_k, \dots, x_n) = \bar{x}_i f(x_1, \dots, 0, \dots, x_n) + x_i f(x_1, \dots, 1, \dots, x_n)$$

As funções  $f(x_1, \dots, 0, \dots, x_n)$  e  $f(x_1, \dots, 1, \dots, x_n)$  são funções de  $n - 1$  variáveis. O teorema pode ser aplicado recursivamente sobre essas duas funções.

**Exemplo:** Consideremos novamente a função  $f(a, b, c, d) = \sum m(2, 5, 8, 9, 11, 12, 14, 15)$ . Vamos mostrar agora como reproduzir a realização mostrada no lado direito da figura 7.12. Como no MUX do segundo nível são usadas as variáveis  $a$  e  $b$ , devemos colocar os produtos que envolvem essas duas variáveis em “evidência”. Em seguida, deve-se fazer o mesmo com a variável  $c$ .

$$\begin{aligned} f &= \bar{a}\bar{b}c\bar{d} + \bar{a}b\bar{c}d + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}d + a\bar{b}cd + a\bar{b}\bar{c}\bar{d} + a\bar{b}c\bar{d} + a\bar{b}cd \\ &= \bar{a}\bar{b}c\bar{d} + \bar{a}b\bar{c}d + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}d + a\bar{b}cd + a\bar{b}c\bar{d} + a\bar{b}cd \quad (\text{rearranjo}) \\ &= \bar{a}\bar{b}c\bar{d} + \bar{a}b\bar{c}d + a\bar{c}\bar{d} + a\bar{b}d + a\bar{b}c \quad (\text{algumas simplificações}) \\ &= \bar{a}(\bar{b}c\bar{d} + b\bar{c}d) + a(\bar{c}\bar{d} + \bar{b}d + bc) \quad (\text{expansão em torno de } a) \\ &= \bar{a}(\bar{b}(c\bar{d}) + b(\bar{c}d)) + a(\bar{b}(\bar{c}\bar{d} + d) + b(\bar{c}\bar{d} + c)) \quad (\text{expansão em torno de } b) \\ &= \bar{a}\bar{b}(c\bar{d}) + \bar{a}b(\bar{c}d) + a\bar{b}(\bar{c}\bar{d} + d) + a\bar{b}(\bar{c}\bar{d} + c) \quad (\text{distribuição com respeito a } a) \\ &= \bar{a}\bar{b}(\bar{c}(0) + c(\bar{d})) + \bar{a}b(\bar{c}(d) + c(0)) + a\bar{b}(\bar{c}(1) + c(d)) + a\bar{b}(\bar{c}(\bar{d}) + c(1)) \quad (\text{expansão em torno de } c) \end{aligned}$$

A última expressão acima corresponde à realização já mostrada anteriormente (na prática, podíamos ter escrito a penúltima linha diretamente a partir da forma canônica inicial).

Nas equações acima, logo após a expansão em torno de  $b$ , poderíamos ter prosseguido da seguinte forma:

$$\begin{aligned} f &= \bar{a}(\bar{b}(c\bar{d}) + b(\bar{c}d)) + a(\bar{b}(\bar{c}\bar{d} + d) + b(\bar{c}\bar{d} + c)) \quad (\text{expansão em torno de } b) \\ &= \bar{a}(\bar{b}c(\bar{d}) + \bar{b}\bar{c}(0) + b\bar{c}(d) + b\bar{c}(0)) + a(\bar{b}\bar{c}(1) + \bar{b}c(d) + b\bar{c}(\bar{d}) + b\bar{c}(1)) \end{aligned}$$

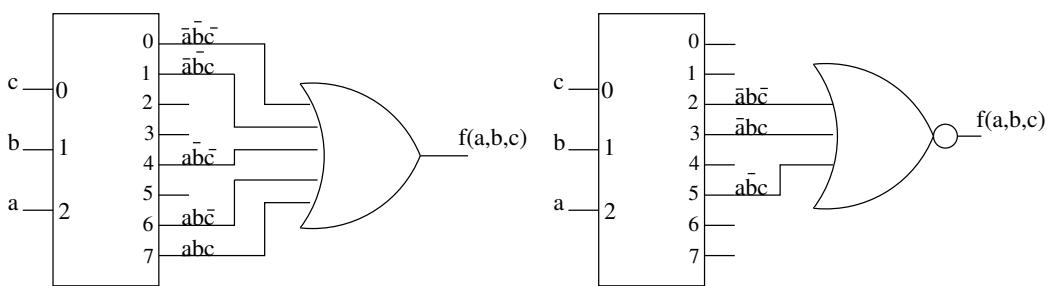
Esta última expressão corresponde à realização mostrada no lado esquerdo da figura 7.12.

### 7.5.3 Realização de funções com decodificadores

Uma vez que um decodificador  $n : 2^n$  realiza todos os produtos canônicos de  $n$  variáveis, qualquer função com  $n$  variáveis pode ser realizada com um decodificador  $n : 2^n$  e uma porta OU (com um número de entradas maior ou igual ao número de 1s da função) ou uma porta NÃO-OU (com um número de entradas maior ou igual ao número de 0s da função).

O custo da realização de uma função com decodificadores é, em termos de portas lógicas, (muito provavelmente) maior que o da realização SOP minimal. No entanto, a simplicidade de projeto torna-o atraente. Além disso, quando múltiplas funções precisam ser realizadas, menor tende a ser a diferença dos custos entre a realização SOP minimal e a realização com decodificadores.

**Exemplo:** A função  $f(a, b, c) = \sum m(0, 1, 4, 6, 7) = \prod M(2, 3, 5)$  pode ser realizada usando decodificadores conforme ilustrado na figura 7.13. No caso da realização com porta NÃO-OU, observe que  $f(a, b, c) = \prod M(2, 3, 5) = \overline{\overline{M_2} \cdot \overline{M_3} \cdot \overline{M_5}} = \overline{\overline{M_2} + \overline{M_3} + \overline{M_5}} = \overline{m_2 + m_3 + m_5}$ .



**Figura 7.13:** Realização de  $f(a, b, c) = \sum m(0, 1, 4, 6, 7)$  com decodificador  $3 : 8$  e uma porta OU (esquerda) ou uma porta NÃO-OU (direita).

### Exercícios

1. Escreva a realização da função  $f(a, b, c) = \bar{a}\bar{b} + a\bar{c}$  usando um MUX  $8 \times 1$ , com as variáveis  $a$ ,  $b$  e  $c$  como seletores.

2. Escreva a realização da função  $f(a, b, c, d) = \sum m(0, 1, 3, 6, 7, 8, 11, 12, 14)$  usando um MUX  $8 \times 1$ , com as variáveis  $a$ ,  $b$  e  $c$  como seletores.
3. Escreva a realização da função  $f(a, b, c, d) = \sum m(0, 1, 3, 6, 7, 8, 11, 12, 14)$  usando um MUX  $4 \times 1$ , com as variáveis  $a$  e  $b$  (neste caso, as entradas possivelmente dependerão das variáveis  $c$  e  $d$  e serão necessárias portas adicionais para a realização de  $f$ ).

# Capítulo 8

## Lógica sequencial – memória

Última atualização em 17 de maio de 2018

Um programa de computador é uma sequência de instruções que podem ser executadas pelo processador. Em geral, enquanto o programa encontra-se em execução, essa sequência de instruções fica armazenada em uma parte do computador denominada memória. A execução dessas instruções pode gerar dados que deverão ser armazenados para utilização posterior, por outras instruções na sequência. Em muitas situações é conveniente que esses dados fiquem também armazenados na memória do computador, para que o posterior acesso a eles seja eficiente. O processador precisa também manter algumas informações que são úteis para controlar a execução dessas instruções. Portanto, memória é um importante componente dos computadores.

Uma característica importante de dispositivos do tipo memória é a capacidade de manter fixo os valores armazenados, exceto quando são explicitamente instruídos a “trocarem” de valor. Dispositivos do tipo memória podem ser modelados por circuitos lógicos. Veremos mais adiante que a capacidade de “armazenar” um certo valor advém da retroalimentação, i.e., fazendo com que o sinal de saída do circuito seja utilizado também como sinal de entrada. A presença de unidades do tipo memória permite que sejam associados aos circuitos a noção de “estado”, caracterizado pelos valores armazenados em suas unidades de memória. A alteração dos valores em suas unidades de memória promove uma transição de estado.

Memória são os elementos que distinguem os circuitos denominados sequenciais daqueles denominados combinacionais.

**Circuito combinacional:** Conforme vimos nos capítulos anteriores, os circuitos combinacionais possuem variáveis de entrada  $x_1, x_2, \dots, x_n$  e uma ou mais saídas  $z_1, z_2, \dots, z_m$  cujos valores são definidos unicamente em função dos valores dessas variáveis de entrada. Isto é, cada uma

das saídas  $z_i$  pode ser expressa por uma função booleana  $f_i : \{0, 1\}^n \rightarrow \{0, 1\}$ .

**Exemplo:** Um típico exemplo de um circuito combinacional é o circuito somador. Ele recebe na entrada a representação binária de dois números e devolve na saída a representação binária da soma dos dois números da entrada. No capítulo anterior vimos outros exemplos de circuitos combinacionais (multiplexador, codificador, etc).

**Circuito sequencial:** Os circuitos sequenciais, por sua vez, também possuem entradas  $x_1, x_2, \dots, x_n$  e uma ou mais saídas  $z_1, z_2, \dots, z_m$ . A diferença fundamental em relação aos circuitos combinacionais é o fato de suas saídas  $z_i$ ,  $i = 1, 2, \dots, m$  dependerem não apenas dos valores das variáveis de entrada, mas também dos valores armazenados nas unidades de memória (i.e., seu “estado atual”).

O estado atual é representado por  $r$  variáveis de estado  $y_1, \dots, y_r$ . Além de funções que definem as saídas do circuito, temos também as funções que definem o “próximo estado” (representado pelas variáveis  $y_1^*, y_2^*, \dots, y_r^*$ ), ou seja,

$$z_i = g_i(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_r), \quad i = 1, 2, \dots, m$$

e

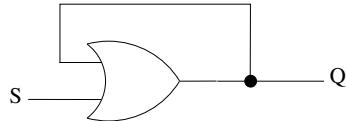
$$y_i^* = h_i(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_r), \quad i = 1, 2, \dots, r$$

Quando consideramos circuitos sequenciais, está implícita a noção de tempo, de iteração. A cada nova iteração, o próximo estado passa a ser o estado atual e um “novo próximo estado” é calculado. Esta relação de realimentação faz com que a interpretação do funcionamento do circuito a partir das equações acima seja não trivial.

**Exemplo:** Os sinais dos semáforos alternam de verde para amarelo e em seguida para vermelho. Depois voltam para verde e este ciclo se repete continuamente. Este é um dispositivo sequencial; isto é, o próximo estado depende do estado atual. Da mesma forma, um elevador é um dispositivo sequencial no qual as entradas são as chamadas feitas (pelas pessoas em cada andar e no interior do elevador) e o estado atual é o andar no qual o elevador se encontra mais o sentido em que ele está se movendo (parado, subindo, descendo). A posição do elevador nos próximos instantes dependerá das entradas atuais e do estado atual dele. Estes exemplos ilustram a ideia de “estado atual” e “próximo estado”.

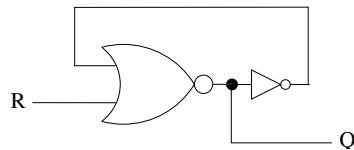
Neste capítulo descrevemos como essas unidades de memória são construídas e funcionam (sob o ponto de vista lógico). Para tanto, vamos examinar inicialmente os circuitos dos dois exemplos a seguir. Eles mostram como a retroalimentação pode ser usada para “congelar” o estado de um circuito. Os circuitos do tipo memória são construídos combinando-se as ideias desses dois exemplos e serão apresentadas após os exemplos.

**Exemplo 1:** Considere uma porta OU com ambas as entradas inicialmente em 0. Nesta situação, a saída é também 0. Suponha que de alguma forma conectamos a saída em uma das entradas (ver figura abaixo). O circuito mantém-se em estado estável. Em seguida, suponha que mudamos o valor da outra entrada (S) para 1. O que acontece com a saída Q? O que acontece se, em seguida, colocamos o valor de S de volta para 0?



Pode-se perceber que quando o valor da entrada S é alterado para 1, a saída do circuito passa a ser 1. Alterações posteriores no circuito não mais afetam a saída. Note também que neste circuito simples o estado e a saída são os mesmos (i.e.,  $z = y$ ).

**Exemplo 2:** Considere uma porta NÃO-OU cuja saída está conectada a um inversor. Este circuito é uma outra forma de representar a função lógica OU. Suponha que inicialmente ambas as entradas da porta NÃO-OU estão em 0. Nesta situação, a saída Q da porta NÃO-OU é 1 e a saída do inversor é 0. Suponha que a saída do inversor é conectada em uma das entradas da porta NÃO-OU (ver figura abaixo). O circuito mantém-se em estado estável. Suponha que em seguida mudamos o valor da outra entrada (R) da porta NÃO-OU para 1. O que acontece com a saída Q? O que acontece se, em seguida, colocamos o valor de R de volta para 0?



De forma similar ao circuito do exemplo anterior, neste caso, quando o valor da entrada R é alterado para 1, o valor do estado muda para 0 e nenhuma alteração posterior no valor de R irá alterar o estado.

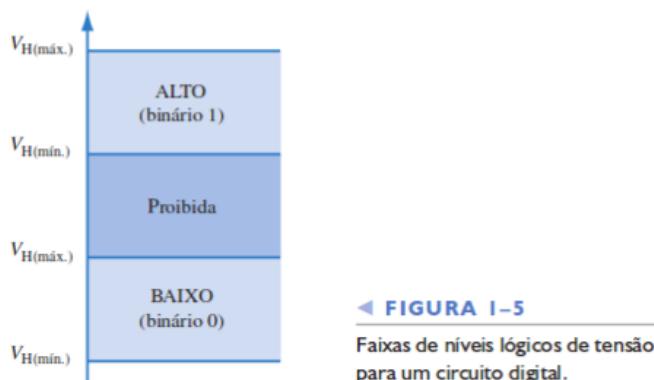
Os exemplos acima são chamados de *set latch* e *reset latch*, respectivamente. No primeiro, após o estado Q passar para 1 (set), ele não pode mais ser alterado. No segundo, após Q passar para 0, não mais pode ser alterado. Portanto, esses são dispositivos que armazenam um certo valor lógico, que representa o estado  $y$ . O estado é denotado por Q nas figuras uma vez que essa é a notação mais comumente encontrada nas referências bibliográficas. Pelo fato de não podermos mudar o estado desses dispositivos mais de uma vez, eles tem possibilidade de uso muito limitado. Podemos, porém, usar idéia similar para construir dispositivos que permitem a alteração de estado múltiplas vezes. Tais circuitos são conhecidos por *set-reset latches* ou *set-reset flip-flops* e descritos mais adiante.

Antes de descrevê-los, convém fazermos uma pausa para introduzir a ideia de ondas digitais que são úteis para entendermos a noção de tempo (ou iteração) em sistemas digitais.

## 8.1 Ondas digitais

Em vez de pensar entradas de um circuito como um valor estático, no contexto de circuitos sequenciais é conveniente pensarmos entradas como um sinal que muda de valor ao longo do tempo. As ondas digitais referem-se a esses sinais. Os conceitos a seguir visam fornecer uma visão geral sobre a representação digital (as figuras que as acompanham foram todas extraídas do livro do Floyd [Floyd, 2007]. Para maiores detalhes veja o capítulo 1 desse livro).

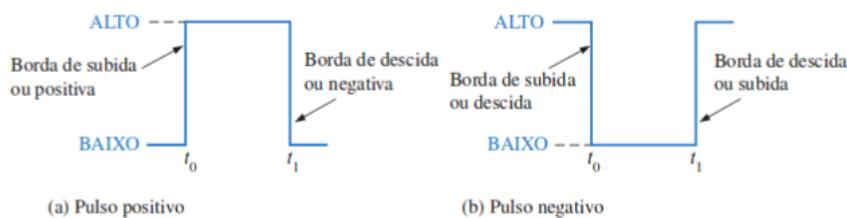
### Interpretação do sinal elétrico como bit



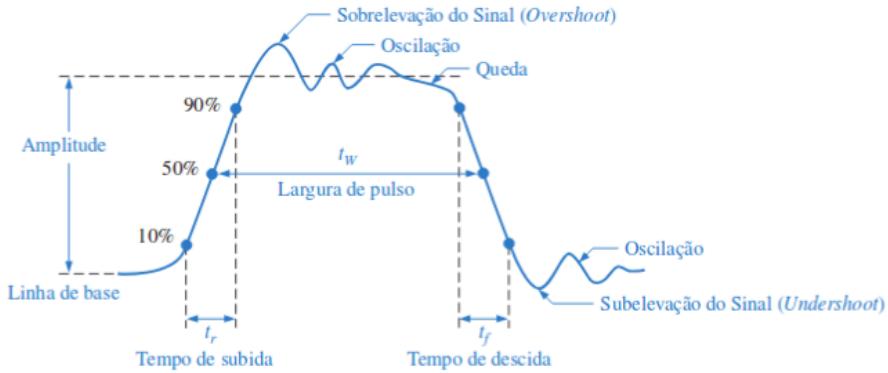
### Onda digital (idealizado)

#### Formas de Onda Digitais

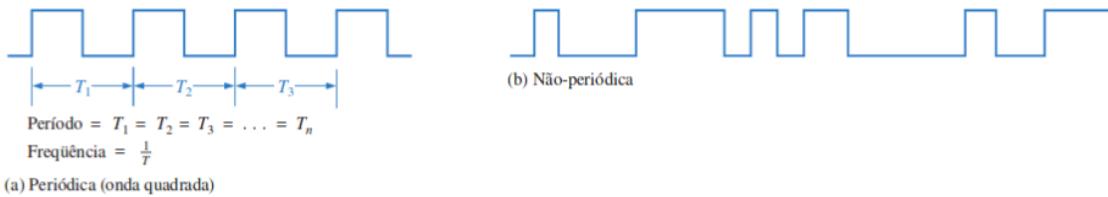
Formas de onda digitais consistem em níveis de tensão que comutam entre os níveis, ou estados, lógicos ALTO e BAIXO. A Figura 1-6(a) mostra que um único pulso positivo é gerado quando a tensão (ou corrente) passa do nível BAIXO normal para o nível ALTO e em seguida retorna para o nível BAIXO. O pulso negativo, visto na Figura 1-6(b), é gerado quando a tensão passa do nível ALTO normal para o nível BAIXO e retorna para o nível ALTO. Uma forma de onda digital é constituída de uma série de pulsos.



Na prática pode haver oscilações



### Sinal periódico e não periódico



Sinal (onda) sincronizado com temporizador pode ser interpretado como uma sequência de bits:

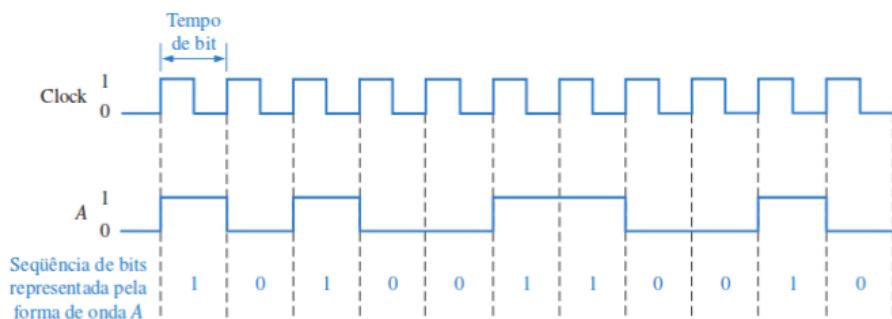
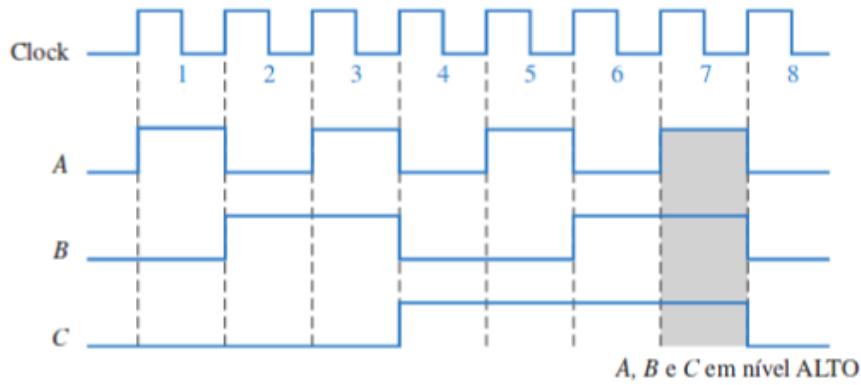
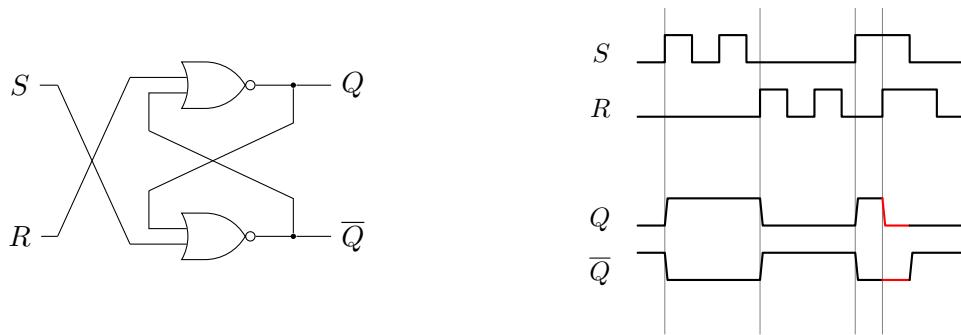


Diagrama temporal: mostrar sinais de forma sincronizada ao temporizador (ou *clock*)



## 8.2 Latches SR

Na figura 8.1, à esquerda encontra-se o circuito de um *latch* SR que usa portas NÃO-OU (NOR), e à direita o seu comportamento (saídas  $Q$  e  $\bar{Q}$ ) em função das entradas ( $S$  e  $R$ ). O nome SR



**Figura 8.1:** Operação de um *latch* SR ( $S$ =sinal set;  $R$ =sinal reset;  $Q$ =saída (estado)), com porta NOR.

vem de *set-reset*. Alguns autores usam a sigla RS em vez de SR.

O diagrama temporal à direita mostra como o estado muda em função de mudanças nas entradas S e/ou R. Inicialmente, ambas as entradas, S e R estão em 0, e a saída Q também está em 0 (consequentemente, a saída  $\bar{Q}$  está em 1). Esse estado é consistente. Então temos a seguinte sequência de eventos:

- o sinal  $S$  vai a 1 e em consequência disso  $\bar{Q}$  vai a 0 e  $Q$  vai a 1.

- o sinal  $S$  vai a 0, mas isso não afeta o estado.
- o sinal  $S$  vai novamente a 1 e em seguida a 0, mas isso também não afeta o estado.
- o sinal  $R$  vai a 1 e em consequência disso  $Q$  vai a 0 e  $\bar{Q}$  vai a 1.
- o sinal  $R$  vai a 0 e em seguida a 1 e logo em seguida a 0 novamente, mas essas variações não afetam o estado.
- o sinal  $S$  vai a 1 e em consequência disso  $\bar{Q}$  vai a 0 e  $Q$  vai a 1.
- o sinal  $R$  vai a 1 e em consequência disso  $Q$  vai a 0, mas  $\bar{Q}$  se matém em 0 (trecho destacado em vermelho). Este é um estado inválido, pois é esperado que  $\bar{Q}$  seja sempre o complementar de  $Q$ .

Podemos verificar que quando a entrada  $S$  vai para 1, o estado  $Q^*$  do *latch* vai para 1 e quando a entrada  $R$  vai a 1, o estado  $Q^*$  do *latch* vai para 0, independentemente do estado atual. Portanto  $S = 1$  indica a operação *set* e  $R = 1$  indica a operação *reset*. Quando ambos estão em 0, nada acontece. Quando ambos vão a 1, i.e.  $S = R = 1$ , temos uma situação que não é previsível. Primeiro, se tivermos ambas as entradas em 1, teremos  $Q = \bar{Q} = 0$ , violando a condição de que uma saída é o complemento da outra. Segundo, ao supormos que  $Q = \bar{Q} = 0$  é aceitável, pode ocorrer uma situação em que ambas as entradas S e R estão em 1 e passam simultaneamente para 0. Em tal situação, a saída das portas NÃO-OU passa a 1 e, em seguida, se as duas portas NÃO-OU funcionarem de forma exatamente iguais, o estado voltará para 0, o que faz com que em seguida passe para 1 e depois novamente para 0 e assim por diante. Isso levaria o estado do circuito a oscilar (ou seja, a saída não se estabilizaria). Se as duas portas não funcionarem de forma exatamente iguais, aquela que responde primeiro ditará o comportamento do circuito. Como na prática é razoável supormos que uma das portas responderá antes da outra e como numa realização física não se tem controle de qual porta responde primeiro, o comportamento será imprevisível.

Pelas razões descritas acima, no *latch SR* a entrada  $R = S = 1$  é proibida. O comportamento do *latch SR* pode ser descrito pela seguinte tabela-verdade:

$S$	$R$	$Q$	$Q^*$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?

Nenhuma mudança  
*reset*  
*set*  
proibido

Para escrever a expressão que representa a relação entrada-saída (ou entrada-próximo estado) desse circuito, considere as seguintes notações:

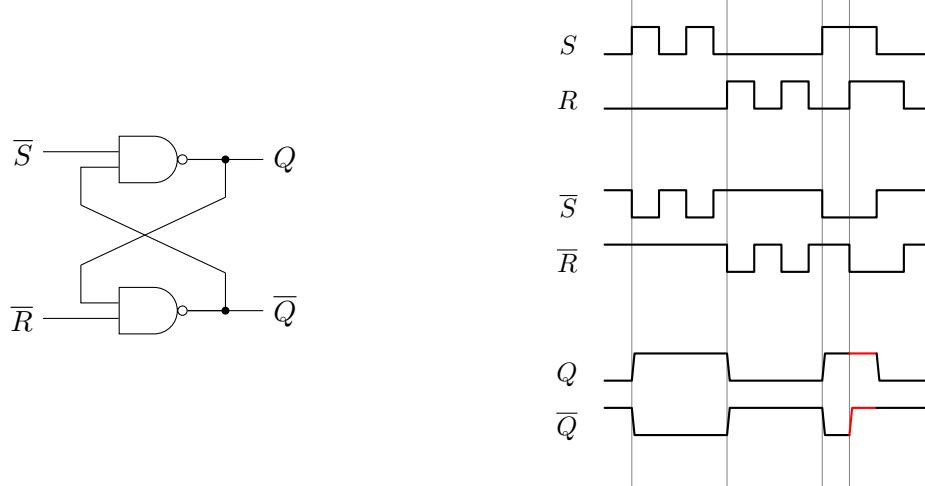
- $S$  denota o valor do sinal que alimenta a entrada  $S$
- $R$  denota o valor do sinal que alimenta a entrada  $R$
- $Q$  denota o estado (e também saída) atual
- $Q^*$  denota o próximo estado.

A situação  $S = R = 1$  é proibida. Portanto podemos tratá-la como *don't care* e então obtemos a expressão que descreve  $Q^*$ :

$$Q^* = S + Q \bar{R}$$

Essa expressão pode ser facilmente obtida desenhando-se o mapa de Karnaugh correspondente à tabela-verdade dada acima.

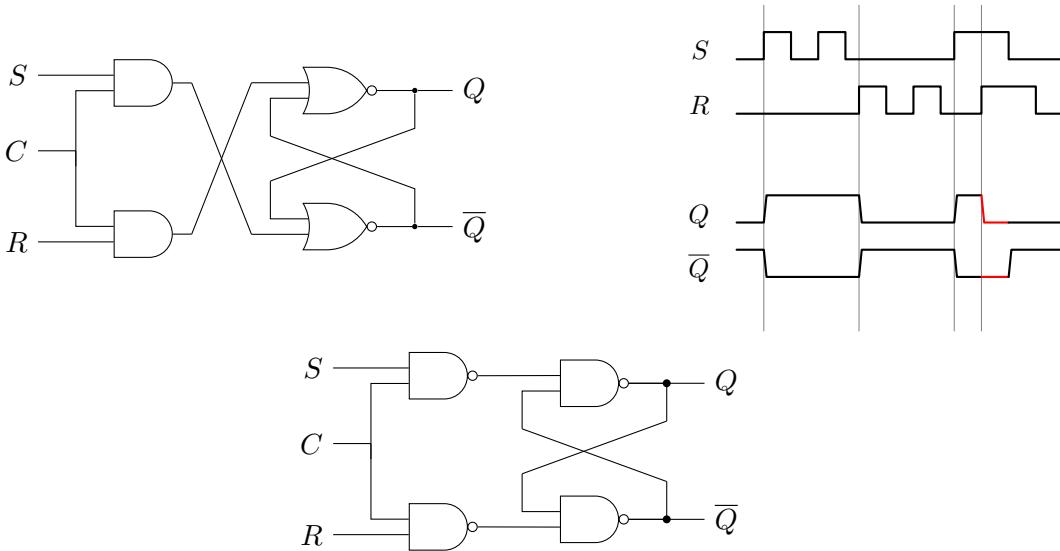
O *latch SR* pode também ser realizado com portas NÃO-E (NAND), da seguinte forma:



**Figura 8.2:** Latch SR ( $S$ =sinal set;  $R$ =sinal reset;  $Q$ =saída (estado)), com porta NAND.

Neste caso também  $S = 1$  faz o estado ir a 1 ( $Q=1$ ). Similarmente,  $R = 1$  faz o estado ir a 0 ( $Q=0$ ). Temos a situação *set* (com  $S = 1$ ) e *reset* ( $R = 1$ ) como no caso anterior. Porém o circuito é tal que a mudança de estado ocorre com o sinal baixo (no caso anterior, ocorria com o sinal alto) e é por esta razão que a primeira entrada é  $\bar{S}$  e a segunda é  $\bar{R}$ . Note que neste caso também  $S = R = 1$  leva o *latch* a um estado inválido.

Cada *latch* pode ser pensado como uma unidade de memória capaz de armazenar um *bit*. Por meio das entradas  $S$  e  $R$  podemos controlar qual valor armazenar. Em geral, um sistema é composto por várias dessas unidades de memória e pode ser interessante termos a possibilidade de controlar quando elas devem mudar de estado. Para isso, pode-se incluir uma entrada de controle como mostrado na figura 8.3. Quando o sinal de entrada  $C$  é 0, a saída de ambas as portas  $E$  é 0 e portanto mudanças no valor de  $R$  e  $S$  não tem efeito nenhum sobre o estado do *latch*. Quando  $C = 1$ , temos o mesmo comportamento descrito anteriormente. A literatura denomina em geral de *latches* quando não temos uma entrada para o sinal de controle e de *flip-flops* quando há. Mas essa nomenclatura parece não ser universal.



**Figura 8.3:** Acima: um *flip-flop SR* (há uma entrada para o sinal de controle), com porta NOR. Abaixo: um *flip-flop SR* com porta NAND. Quando  $C = 1$ , estes *flip-flops* comportam-se da mesma forma que o *latch SR* mostrado na figura 8.1.

Um sinal do *clock* pode ser usado para sincronizar a mudança de estados de múltiplos *flip-flops*. Note que é importante garantir que os sinais na entrada  $S$  e  $R$  estejam devidamente preparados no momento em que o sinal  $C$  sobe, para que as mudanças esperadas ocorram.

Daqui em diante, sempre que mencionarmos *flip-flop SR*, estaremos considerando a existência de entrada para o sinal de controle. Considerando a entrada  $C$  para o sinal de controle, a expressão do próximo estado de um *flip-flop SR* é dada por

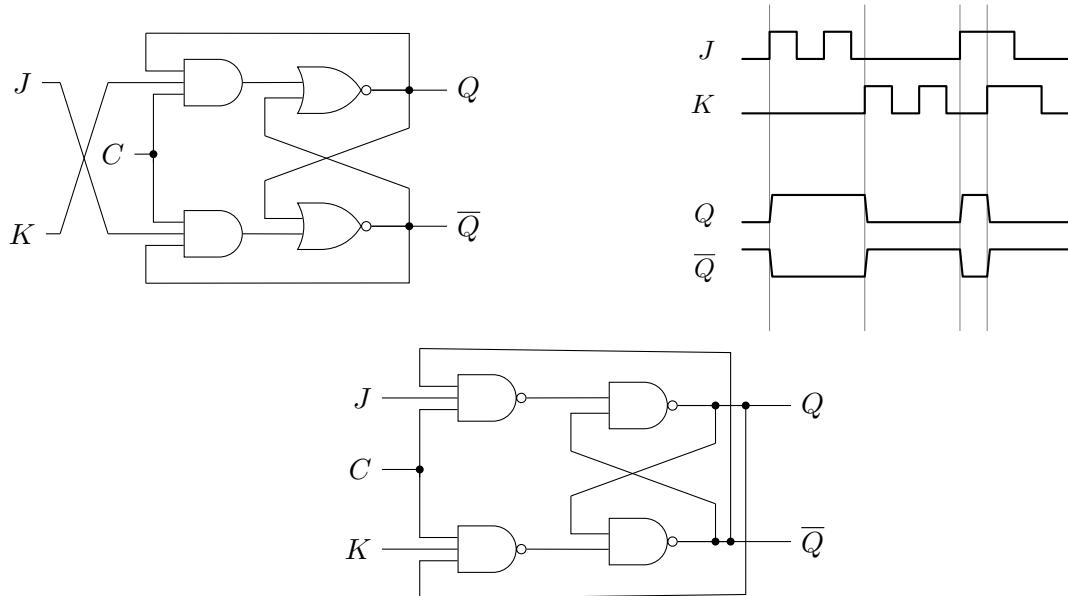
$$Q^* = S C + Q \bar{R} + Q \bar{C}.$$

Se  $C = 0$ , temos  $Q^* = Q$ ; se  $C = 1$  então temos  $Q^* = S + Q \bar{R}$ , que é a mesma equação do *latch SR*.

### 8.3 *Flip-flop JK*

Vimos que nos *flip-flops* SR, se ambas as entradas estiverem em 1 simultaneamente, eles podem apresentar comportamento imprevisível. Isso, do ponto de vista prático, é indesejável pois o projetista do circuito precisaria garantir que as duas entradas nunca ficarão ativas simultaneamente.

Os *flip-flops* JK são uma evolução do SR pois não apresentam esse problema. No JK, quando ambas as entradas ficam em 1 simultaneamente, o estado do circuito se inverte, isto é, se era 1 vai a 0 e vice-versa. A figura a seguir mostra duas realizações do *flip-flop* JK, uma usando portas NOR e outra usando portas NAND. Ambas comportam-se da mesma forma. Note que quando  $J = K = 1$ , no mesmo ponto no qual o *flip-flop* SR gerava um estado inválido, ocorre a inversão de estado. É importante notar que o sinal em  $C$  deve ser um pulso (valor 1 por um pequeno instante de tempo) para que a saída das portas no primeiro nível não mude mais de uma vez. Isto é, quando uma eventual alteração em  $Q$  ou  $\bar{Q}$  chegar a essas portas,  $C$  já deve estar em 0. Caso  $C$  esteja em 1, quando  $J = K = 1$ , uma nova rodada de propagação de sinais ocorrerá e o *flip-flop* poderá apresentar comportamento não esperado.



**Figura 8.4:** Flip-flop J-K, com portas NOR e com portas NAND.

Portanto, o comportamento do *flip-flop* JK pode ser resumido pela seguinte tabela-verdade:

$J$	$K$	$C$	$Q^*$	
$\times$	$\times$	0	$Q$	não muda
0	0	1	$Q$	mantém
0	1	1	0	reset
1	0	1	1	set
1	1	1	$\bar{Q}$	inverte

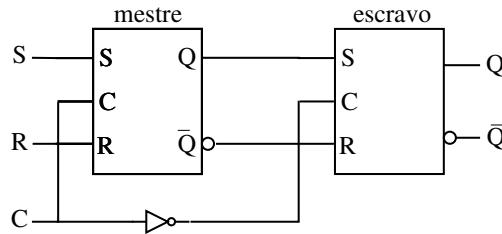
Para  $C = 1$  a equação do próximo estado é dada por

$$Q^* = J\bar{Q} + \bar{K}Q.$$

## 8.4 *Flip-flops* mestre-escravo

Dependendo do período de um pulso do *clock*, em um *flip-flop* JK pode ocorrer oscilação em seu estado quando  $J = K = 1$  (por exemplo, o estado pode mudar duas vezes em vez de mudar apenas uma vez).

Para contornar o problema acima, foi desenvolvido o ***flip-flop* mestre-escravo**. A característica deles é o fato de mudarem de estado uma única vez a cada pulso, independentemente da largura dele. Para entender esse mecanismo, usaremos o esquema de um *flip-flop* SR mestre-escravo, mostrado na figura a seguir.

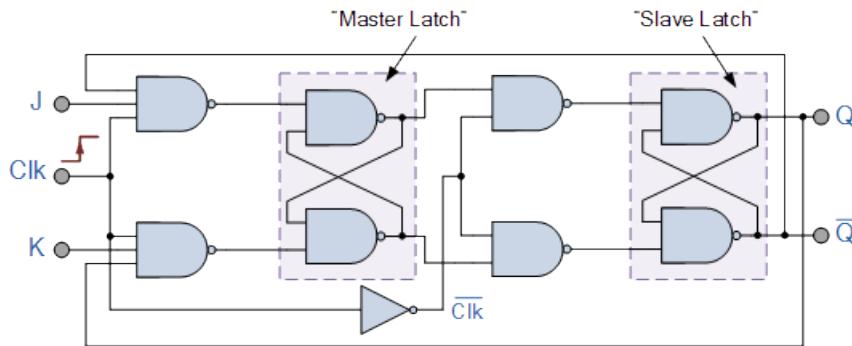


Quando o *clock* está em 0, o primeiro deles (mestre) está desabilitado para mudanças. O segundo está habilitado para mudanças, mas como o primeiro não varia, o seu estado é uma cópia do estado do primeiro. Quando o *clock* está em 1, o estado do primeiro varia de acordo com as variações nas entradas  $S$  e  $R$ , mas o segundo está “congelado” e portanto seu estado não varia.

Quando o *clock* sobe, o mestre pode mudar de estado de acordo com as entradas  $S$  e  $R$  enquanto o escravo fica desabilitado (deve-se apenas garantir que o escravo fique desabilitado antes que ocorra qualquer mudança na saída do mestre; isso é garantido pelo “delay” da porta NÃO). Quando o *clock* desce, o mestre fica desabilitado (e portanto “congela” a saída dele) e a mudança de estado no escravo fica habilitada (assim, a saída do mestre é copiada para a saída do escravo). Esse mecanismo garante que a cada pulso ocorrerá apenas uma mudança de estado. Obviamente,

se as entradas variarem enquanto o *clock* estiver alto, o estado do mestre variará de acordo. Em todo o caso, é necessário garantir que as estradas estão estabilizadas quando a mudança de estados é ativada.

No caso dos *flip-flops* JK, certamente a oscilação nas entradas irá provocar oscilação no estado. Porém a oscilação crítica que motivou a criação do esquema mestre escravo é a oscilação que pode ocorrer em situações nas quais  $J = K = 1$  e o sinal de controle fica alto por um tempo maior que o ideal. A figura 8.5 mostra o *flip-flop* JR mestre-escravo:

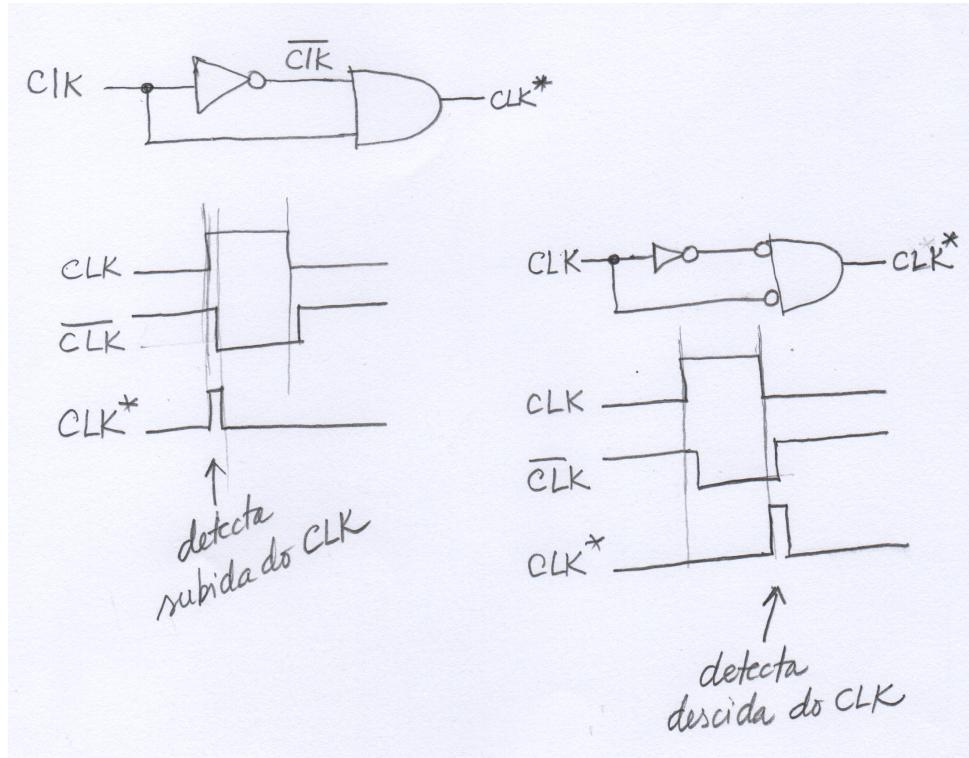


**Figura 8.5:** Esquema de um flip-flop JK mestre-escravo (imagem extraída de [https://www.electronics-tutorials.ws/sequential/seq\\_2.html](https://www.electronics-tutorials.ws/sequential/seq_2.html))

Na situação  $J = K = 1$  e  $C = 1$ , as portas NAND no primeiro nível garantem que as entradas  $S$  e  $R$  do *latch* mestre nunca serão tais que  $S = R = 1$  (pois uma porta NAND é alimentado pelo  $Q$  e o outro pelo  $\bar{Q}$ ). Isto irá evitar que ocorra a oscilação decorrente do fato das entradas serem  $J = K = 1$ . Desde que essas entradas sejam mantidas estabilizadas enquanto  $C$  está em 1, garante-se que haverá apenas uma mudança de estado a cada pulso.

Atualmente os *flip-flops* mestre-escravo são considerados um tanto obsoletos. O problema da oscilação quando  $J = K = 1$  foi contornado por meio dos detectores de borda. A ideia central dos detectores de borda é identificar as bordas de subida ou de descida de um pulso. Como vimos acima, no caso do *flip-flop* JK, se o pulso for largo quando  $J = K = 1$  o estado irá oscilar. Ao se usar um detector de borda de subida, um pulso é transformado em um pulso de curtíssima duração, suficiente para ativar a mudança e permitir uma, mas não mais que uma, mudança de estado.

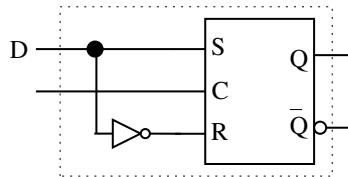
Os *flip-flops* com esse mecanismo de detecção de bordas são chamados de *flip-flops* ativados por borda (*edge triggered flip-flops*).



**Figura 8.6:** Circuitos para detecção de borda de subida e descida, aproveitando o atraso da porta NÃO.

## 8.5 Flip-flops D e T

**Flip-flop D:** trata-se de *flip-flop* que armazena o valor  $D$  que está na entrada sempre que há um pulso na entrada  $C$ . Um *flip-flop* desses pode ser implementado usando um *flip-flop* SR, alimentando a entrada  $S$  com o sinal  $D$  e a entrada  $R$  com o seu complemento (isso garante também que  $S$  e  $R$  nunca estarão em 1 simultaneamente), como na figura a seguir. Pode-se também usar um *flip-flop* JK em vez de um *flip-flop* SR.

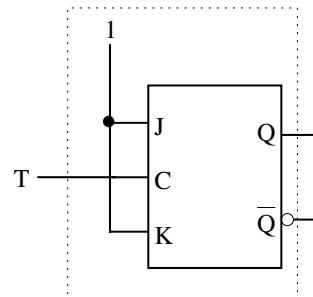


A expressão do próximo estado do *flip-flop* D é dada por

$$Q^* = D C + \overline{C} Q$$

ou seja, se  $C = 0$ ,  $Q^* = Q$  e, se  $C = 1$ , então  $Q^* = D$ .

**Flip-flop T:** trata-se de um *flip-flop* que, quando há um pulso na entrada  $T$ , inverte o seu estado. Um *flip-flop* desses pode ser implementado usando um *flip-flop* JK, com ambas as entradas,  $J$  e  $K$ , fixas em 1 e o sinal  $T$  alimentando a entrada de controle  $C$ , como na figura a seguir.



A expressão do *flip-flop* T é dada por

$$Q^* = \overline{Q}.$$

# Capítulo 9

## Circuitos sequenciais – Registradores e contadores

Última atualização em 17 de maio de 2018

### 9.1 Síncronos × Assíncronos

Em **circuitos síncronos**, a mudança de estado de todos os elementos de memória do circuito (*flip-flops*) ocorre em sincronia (“ao mesmo tempo”) e é controlada pelo sinal de um *clock*.

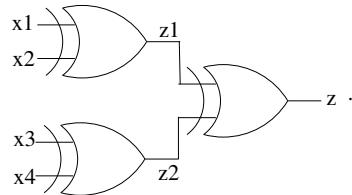
Em **circuitos assíncronos**, a mudança de estado não é sincronizada, ou seja, não se utiliza o sinal de um *clock* para se promover a mudança de estado de todos os *flip-flops*. Há dois modos de promover a mudança de estado em circuitos assíncronos. No modo pulso, o controle do *flip-flop* pode ser alimentado por qualquer outro sinal. No modo fundamental, o retardo intrínseco ou propositalmente colocado no circuito é utilizado para funcionar como “memória”. Em ambos os casos há restrições que devem ser satisfeitas para o circuito funcionar propriamente.

Nesta seção apresentamos alguns poucos exemplos de circuitos sequenciais. A maior parte dos exemplos que veremos serão de circuitos síncronos. Quando não for o caso, isso será explicitamente mencionado.

## 9.2 Verificador de paridade em transmissão serial

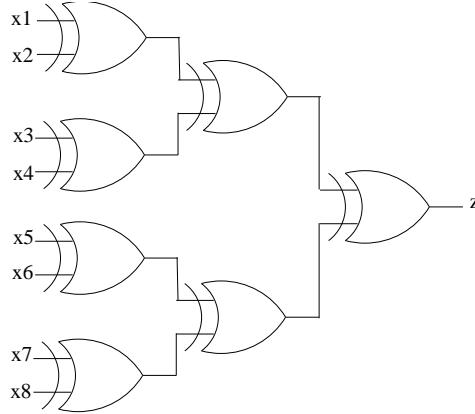
Suponha que, em um sistema de transmissão, os dados são agrupados de 7 em 7 bits e que, para cada grupo de 7 bits, um oitavo bit com a informação de paridade é adicionado. O valor desse oitavo bit depende dos valores dos 7 primeiros bits. Se o número de bits 1 nos 7 bits é ímpar, o oitavo bit é 1; se é par, o oitavo bit é 0. Assim, a cada grupo de oito bits transmitidos tem-se necessariamente um número par de bits 1. Após a transmissão dos dados, o receptor pode verificar a paridade de cada 8 bits e detectar erros (não todos) que possam ter ocorrido no trajeto da transmissão.

A figura 9.1 mostra um circuito verificador de paridade para 4 bits, baseado em portas XOR. A saída desse circuito é 1 se, e somente se, a paridade é ímpar.



**Figura 9.1:** Circuito para verificação de paridade para uma entrada de 4 bits.

O circuito acima pode ser estendido para verificar a paridade de 8 bits, simplesmente conectando-se a saída de dois circuitos daqueles a uma porta XOR, conforme mostrado na figura 9.2.



**Figura 9.2:** Circuito para verificação de paridade para entradas de 8 bits.

Caso a transmissão seja paralela (isto é, os 8 bits são transmitidos simultaneamente por 8 linhas de transmissão), a verificação de paridade poderia ser efetuada imediatamente na chegada do sinal. No entanto, se a transmissão for serial (*bits* transmitidos sequencialmente através de uma única linha de transmissão), o circuito verificador de paridade teria de aguardar a chegada dos

8 bits para gerar o resultado da verificação. Em muitas situações, a transmissão é feita no modo serial.

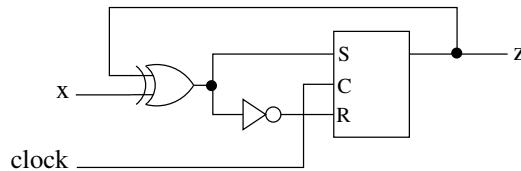
Observe que a expressão da saída  $z$  do circuito acima é dada por

$$z = [(x_1 \oplus x_2) \oplus (x_3 \oplus x_4)] \oplus [(x_5 \oplus x_6) \oplus (x_7 \oplus x_8)]$$

mas ela pode também ser expressa por

$$z = \left[ \left( \left[ \left( (x_1 \oplus x_2) \oplus x_3 \right) \oplus x_4 \right) \oplus x_5 \right] \oplus x_6 \right] \oplus x_7 \oplus x_8 .$$

Esta última equação sugere que a verificação de paridade pode ser realizada iterativamente, bit a bit. O circuito sequencial que realiza tal verificação é mostrado na figura 9.3. Suponha que o flip-flop está inicialmente em 0.



**Figura 9.3:** Circuito sequencial para verificação de paridade.

Nesse circuito, assim que o oitavo bit chega ao destino e passa pelo circuito, tem-se a verificação de paridade concluída. Este exemplo, além de ser um exemplo que ilustra o uso de flip-flops, é também um exemplo que ilustra situações nas quais um circuito sequencial faz mais sentido do que um combinacional. Além disso, esse mesmo verificador de paridade sequencial pode ser usado independentemente do número de bits considerado; já o combinacional tem número de entradas fixa.

### 9.3 Registradores

Ver, por exemplo, o capítulo 9 do Floyd.

### 9.4 Contadores

O objetivo de um circuito contador é, a cada pulso do sinal de *clock*, incrementar (ou decrementar) o valor armazenado em alguma memória.

Um **contador incremental módulo  $2^n$** , com valor inicial 0, apresenta a seguinte sequência de

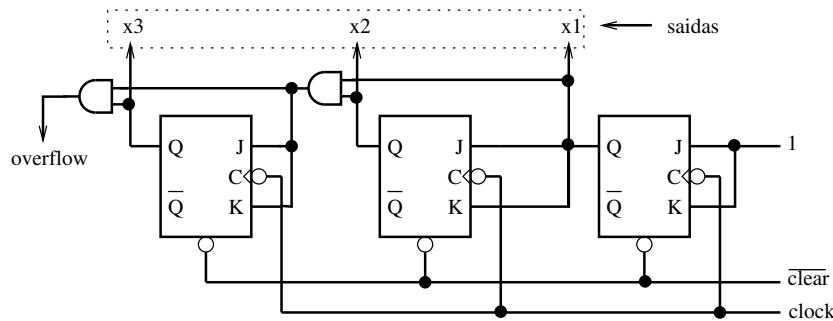
transição de valores:

$$0, 1, 2, 3, \dots, 2^n - 1, 0, 1, 2, 3, \dots$$

**Exemplo:** Um contador incremental módulo  $2^3$  é definido pela seguinte tabela:

Estado atual $x_3\ x_2\ x_1$	Próximo estado $x_3^*\ x_2^*\ x_1^*$		
	$x_3^*$	$x_2^*$	$x_1^*$
000	0	0	1
001	0	1	0
010	0	1	1
011	1	0	0
100	1	0	1
101	1	1	0
110	1	1	1
111	0	0	0

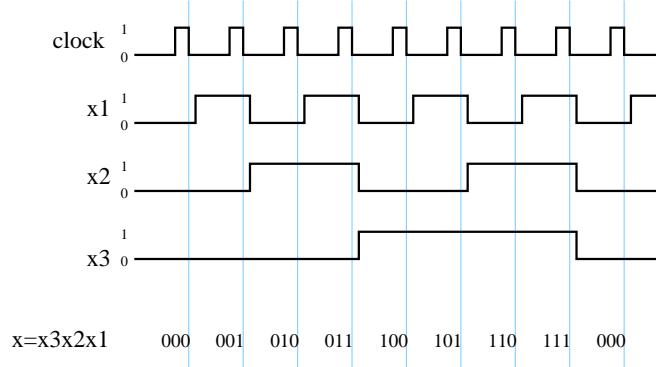
Uma possível implementação de um contador incremental módulo  $2^3$ , usando *flip-flops JK edge-triggered*, é mostrado na figura a seguir:



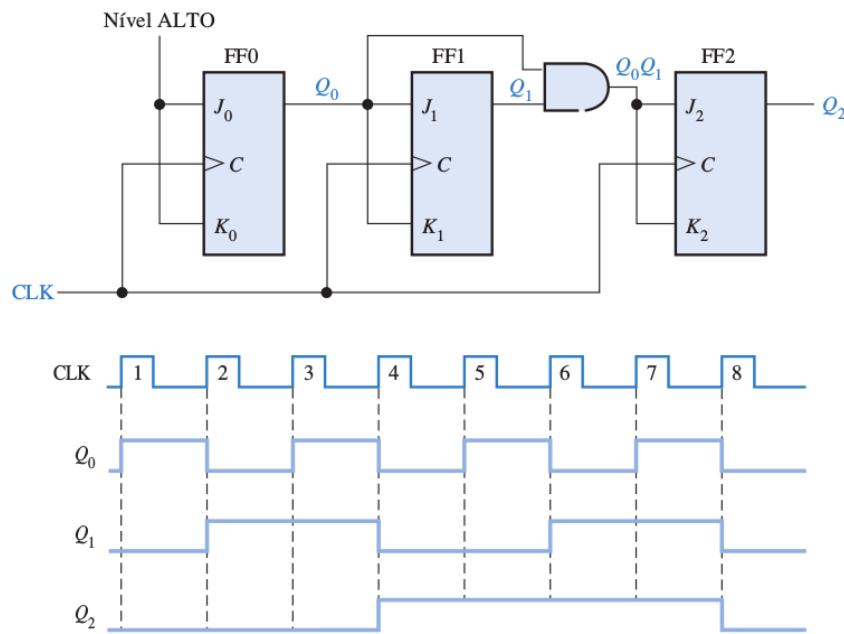
Observe que nos três *flip-flops* temos  $J = K$ . Os *flip-flops* usados são *edge-triggered*, com a mudança de estado ocorrendo na descida do sinal de *clock*. O valor armazenado num certo instante é dado por  $x = x_3 x_2 x_1$ . O sinal *clear* indica que os *flip-flops* são reinicializados com sinal baixo (portanto quando *clear* = 1).

Suponha que inicialmente um pulso no sinal *clear* “zera” todos os *flip-flops* e portanto o valor armazenado é 000. A cada pulso do *clock*, o estado do *flip-flop* mais à direita inverte (passa de 0 para 1, ou de 1 para 0). O estado do segundo *flip-flop* muda a cada dois pulsos do *clock*: mais precisamente, no início é 0 e no primeiro pulso do *clock* também permanece em 0 pois  $x_1$  é 0. No segundo pulso, como  $x_1$  é 1, inverte o estado, com  $x_2$  passando para 1. No terceiro pulso, como  $x_1$  é 0,  $x_2$  não muda e permanece em 1. No quarto pulso,  $x_1$  é 1 e portanto inverte a saída do segundo *flip-flop*, ou seja,  $x_2$  volta a 0, e assim por diante. A mudança de estado

do terceiro *flip-flop* (e portanto de  $x_3$ ) ocorre de forma similar ao do segundo *flip-flop*, porém a cada 4 pulsos do *clock*. O bit *overflow* é 1 quando todos os bits  $x_i$  são 1. Quando o circuito encontra-se no estado 111, o próximo estado será 000. A simulação do circuito pode ser vista na figura a seguir.



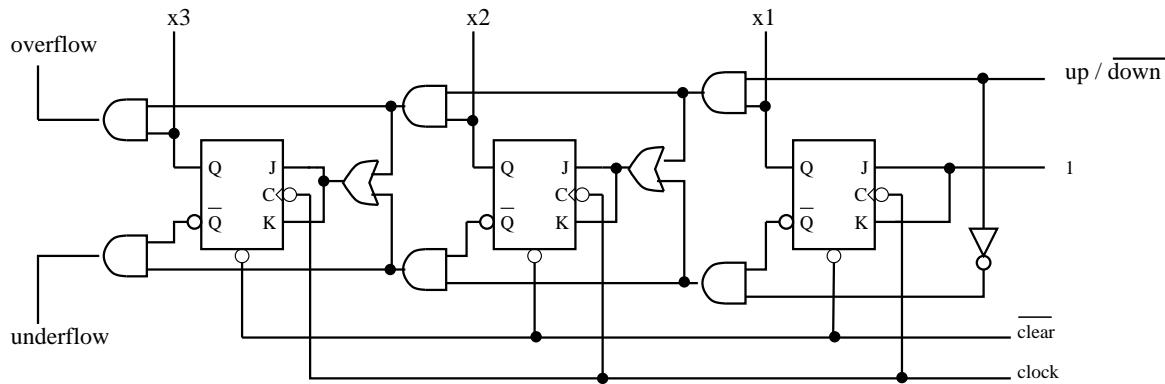
Outra implementação pode ser realizada com flip-flops ativados na borda de subida do clock.



Floyd, Sistemas Digitais – Fundamentos e Aplicações, Pág. 454, edição 9

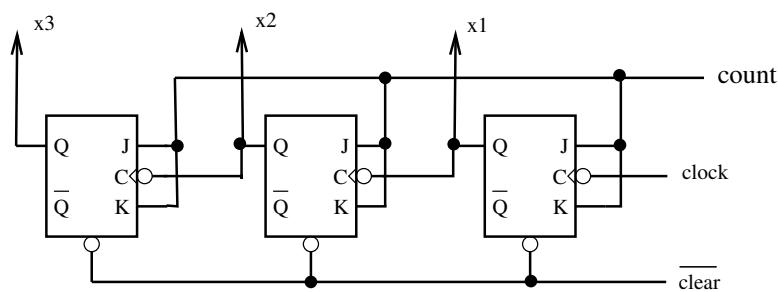
Contadores incremental/decremental (que ora operam incrementalmente e ora operam decrementalmente) podem ser obtidos modificando-se ligeiramente a estrutura do contador incremen-

tal acima. Haverá um sinal de entrada adicional para controlar o modo de operação (incremento/decremento) do contador, conforme figura a seguir.



Observe que se o sinal *up* estiver alto, o comportamento do circuito será exatamente igual ao do circuito anterior. Se o sinal *up* estiver baixo, então o circuito realiza a operação de decremento, de forma muito similar à operação de incremento. Simule o circuito para ver a sequência de *bits* gerados pelo circuito quando *up* = 0.

Podemos também ter contadores incrementais módulo  $2^n$  assíncronos, ou seja, aqueles cuja mudança de estado de todos os *flip-flops* não são controlados pelo sinal *clock*, como no circuito a seguir.



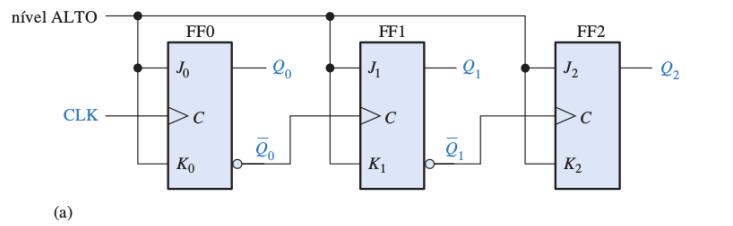
O circuito acima é baseado na observação de que, num contador incremental, ocorre inversão de um certo *bit*  $x_i$  sempre que há uma transição de 1 para 0 no *bit*  $x_{i-1}$ . Veja isso na tabela a seguir:

<b>x</b>	$x_3$	$x_2$	$x_1$
0	0	0	0
1	0	0	1
			↓
2	0	1	0
3	0	1	1
			↓ ↓
4	1	0	0
5	1	0	1
			↓
6	1	1	0
7	1	1	1
			↓ ↓
0	0	0	0

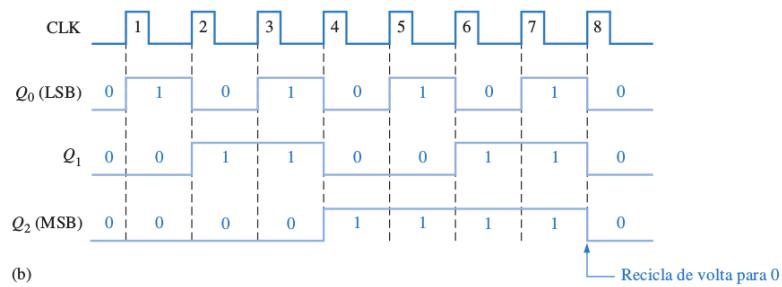
O sinal *count*, quando alto, habilita o circuito para contagem, enquanto que se baixo desabilita a contagem (fazendo com que o circuito funcione como uma memória – *data holder*). Supondo  $count = 1$  e o valor inicial armazenado igual a 000, a cada descida do sinal de *clock* o estado do *flip-flop* mais à direita é invertido (como no circuito anterior). Apenas o primeiro *flip-flop* é alimentado pelo sinal do *clock*. Os demais *flip-flops* são alimentados pelas saídas dos *flip-flops* anteriores. Toda vez que há uma transição de 0 para 1 na saída de um *flip-flop*, a saída do próximo *flip-flop* é invertida.

Devido aos atrasos intrínsecos na propagação de sinal em um *flip-flop*, podem ocorrer estados transitórios na passagem de um estado para outro do circuito. Por exemplo, do estado 011 o circuito passa transitoriamente pelos estados 010 e 000 até ficar em 100 (que seria o estado seguinte ao estado 011). Portanto, circuitos combinacionais que possam fazer uso dos valores  $x_3x_2x_1$  devem ser projetados de forma a evitar esses estados transitórios. Tal efeito (estados transitórios) pode ser verificado fazendo-se uma simulação do circuito e considerando que há um certo retardo até a saída de um *flip-flop* afetar a saída do próximo *flip-flop*.

Simulação de um contador assíncrono de 3 bits.



(a)



Floyd, Sistemas Digitais – Fundamentos e Aplicações, Pág. 446, edição 9

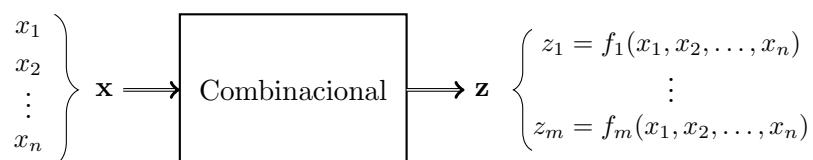
# Capítulo 10

## Análise e projeto de circuitos sequenciais

Última atualização em 13 de junho de 2018

Neste capítulo o objetivo é a análise e projeto de circuitos sequenciais síncronos, aqueles nos quais todos os elementos de memória – i.e., *flip-flops* – mudam de estado de forma sincronizada com um pulso de *clock*. Inicialmente, na parte de análise, examinaremos como proceder para se entender o funcionamento desses circuitos, por um método que não seja apenas por via de simulações e seus diagramas temporais. Em seguida, na parte de projeto, estudaremos como projetar circuitos desse tipo a partir de uma descrição funcional. Para isso, é conveniente recordamos a diferença entre circuitos combinacionais e sequenciais e traçar um paralelo de análise e projeto entre circuitos dessas duas famílias.

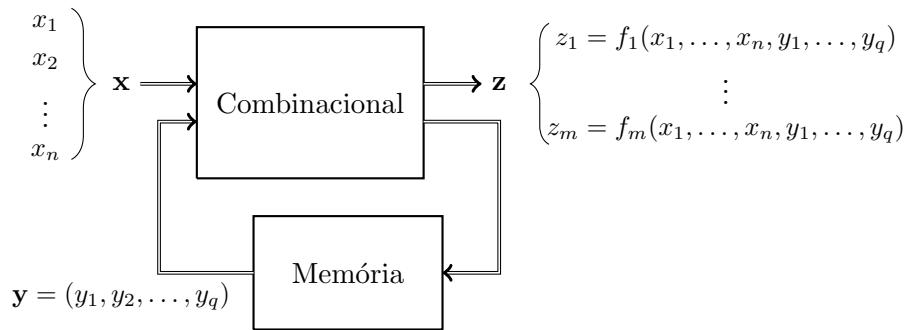
Circuitos combinacionais são aqueles que possuem um certo número  $n$  de entradas e  $m$  de saídas, tal que as saídas dependem apenas das entradas. O diagrama da figura 10.1 mostra a estrutura geral de circuitos combinacionais.



**Figura 10.1:** Estrutura de um circuito combinacional:  $n$  entradas denotadas  $x_1, \dots, x_n$  e  $m$  saídas denotadas  $z_1, \dots, z_m$ , com  $z_i = f_i(x_1, \dots, x_n)$ .

Já os circuitos sequenciais são aqueles que incluem memória. A figura 10.2 mostra um diagrama com a estrutura geral de circuitos sequenciais. Além de uma parte combinacional, há a memória

que consiste de um conjunto de *flip-flops*. Note que tanto as saídas  $\mathbf{z}$  como o próximo estado  $\mathbf{y}^*$  dependem das entradas  $\mathbf{x}$  e do estado atual  $\mathbf{y}$ .



**Figura 10.2:** Estrutura de um circuito sequencial:  $n$  entradas denotadas  $x_1, \dots, x_n$ ,  $m$  saídas denotadas  $z_1, \dots, z_m$  e  $q$  estados denotados  $y_1, \dots, y_q$ , com  $z_i = f_i(x_1, \dots, x_n, y_1, \dots, y_q)$ .

No caso de circuitos combinacionais, com relação ao processo de análise vimos que a função correspondente ao circuito pode ser expressa algebraicamente de forma direta a partir do desenho do circuito. A partir da expressão booleana, pode-se então gerar a tabela-verdade que define de forma explícita a relação entrada-saída. Por outro lado, o projeto de circuitos pode ser abordado a partir da especificação da relação entrada-saída dada pela tabela-verdade. O procedimento comum é a construção da expressão na forma canônica, seguida de sua minimização, e “tradução” direta da expressão simplificada para o desenho de um circuito.

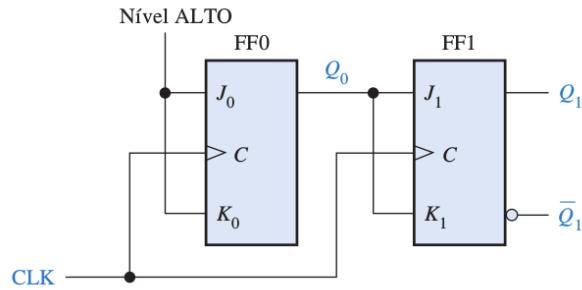
No caso de circuitos sequenciais, a ideia é a mesma. No processo de análise, inicialmente determinamos as equações que definem a entrada de cada um dos *flip-flops*, e em função delas a equação do próximo estado. Essas equações são suficientes para descrevermos a transição de estados, que quando mostradas de forma visual por meio de um diagrama permitem um entendimento do funcionamento do circuito. No processo de projeto, segue-se o caminho inverso, isto é, parte-se de uma descrição funcional (uma descrição sobre como deve funcionar o circuito) para se chegar ao circuito propriamente. Uma descrição funcional pode ser dada, por exemplo, por meio de um diagrama de estados e a partir deste pode-se determinar as transições de estados e as entradas em cada um dos *flip-flops* para que tais transições ocorram. Tendo-se essas equações, pode-se desenhar o circuito. Devido ao conceito de estado, o projeto de circuitos sequenciais não é tão direto como o de circuitos combinacionais. Os processos de análise e projeto serão examinados em detalhes a seguir por meio de exemplos.

## 10.1 Análise de circuitos sequenciais

No processo de análise, dado o desenho de um circuito, desejamos entender o seu funcionamento. Em outras palavras, queremos saber para que ele serve. O processo envolve os seguintes passos:

1. Determinar a equação das entradas de cada *flip-flop* e também das saídas do circuito
2. Determinar a equação do próximo estado de cada *flip-flop*
3. Determinar a tabela de transição de estados
4. Desenhar o diagrama de transição de estados

**Exemplo 1:** Circuito do contador (de 2 bits) módulo 4



**Figura 10.3:** Contador síncrono de 2 bits.

1. Equação das entradas dos *flip-flops*

$$J_0 = K_0 = 1$$

$$J_1 = K_1 = Q_0$$

2. Equação do próximo estado (Lembre que  $Q^* = J\bar{Q} + \bar{K}Q$ )

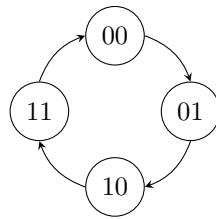
$$Q_0^* = \bar{Q}_0 \quad (\text{já que } J_0 = K_0 = 1)$$

$$Q_1^* = J_1 \bar{Q}_1 + \bar{K}_1 Q_1 = Q_0 \bar{Q}_1 + \bar{Q}_0 Q_1 = Q_0 \oplus Q_1$$

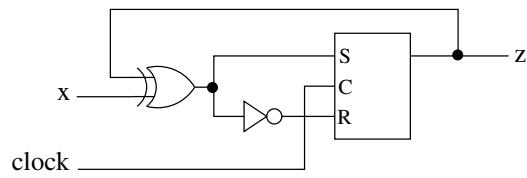
3. Tabela de transição de estados: baseado nas equações do item anterior,

Estado atual ( $Q_1 Q_0$ )	Próximo estado $Q_1^* Q_0^*$
00	01
01	10
10	11
11	00

4. Diagrama de transição de estados



**Exemplo 2:** Circuito verificador de paridade



1. Equação das entradas dos *flip-flops*

$$S = Q \oplus x$$

$$R = \bar{S}$$

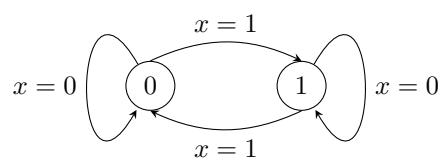
2. Equação do próximo estado (lembre que  $Q^* = S + Q\bar{R}$ )

$$Q^* = S + Q\bar{R} = S + Q\bar{\bar{S}} = S = Q \oplus x$$

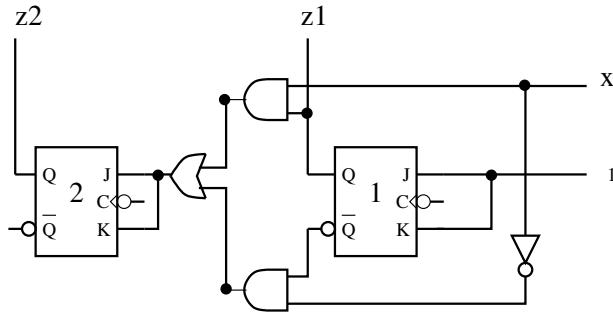
3. Tabela de transição de estados

Estado atual (Q)	Próximo estado ( $Q^*$ )	
	$x = 0$	$x = 1$
0	0	1
1	1	0

4. Diagrama de transição de estados



**Exemplo 3:** Circuito contador *up-down*



[Refazer esta figura](#)

- Equação das entradas dos *flip-flops*

$$J_1 = K_1 = 1$$

$$J_2 = K_2 = x Q_1 + \bar{x} \bar{Q}_1$$

- Equação da saída e do próximo estado

$$z_1 = Q_1$$

$$z_2 = Q_2$$

$$Q_1^* = 1 \bar{Q}_1 + \bar{1} Q_1 = \bar{Q}_1$$

$$Q_2^* = (x Q_1 + \bar{x} \bar{Q}_1) \bar{Q}_2 + (x Q_1 + \bar{x} \bar{Q}_1) Q_2$$

$$Q_2^* = \begin{cases} \bar{Q}_1 \bar{Q}_2 + Q_1 Q_2, & \text{se } x = 0, \\ Q_1 \bar{Q}_2 + \bar{Q}_1 Q_2, & \text{se } x = 1. \end{cases}$$

- Tabela de transição de estados: as saídas foram omitidas na tabela já que  $z_1 = Q_1$  e  $z_2 = Q_2$

Estado atual ( $Q_1 Q_2$ )	$Q_1^* Q_2^*$	
	$x = 0$	$x = 1$
00	11	10
01	10	11
10	00	01
11	01	00

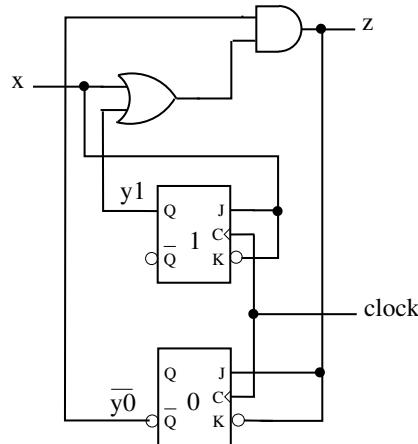
Vamos agora colocar o estado na forma  $Q_2 Q_1$ :

Estado atual ( $Q_2 Q_1$ )	$Q_2^* Q_1^*$	
	$x = 0$	$x = 1$
00	11	01
01	00	10
10	01	11
11	10	00

#### 4. Diagrama de transição de estados

Desenhe o diagrama de estados

#### Exemplo 4:



(notação usada aqui:  $y_0 = Q_0$  e  $y_1 = Q_1$ )

#### 1. Equação das entradas dos *flip-flops*

$$J_1 = x$$

$$K_1 = \bar{x}$$

$$J_0 = z = (x + Q_1) \bar{Q}_0$$

$$K_0 = \bar{z} = (x + Q_1) \bar{Q}_0$$

#### 2. Equação da saída e dos próximos estados: (Lembre que $Q^* = J\bar{Q} + \bar{K}Q$ )

$$z = (x + Q_1) \bar{Q}_0$$

$$Q_0^* = [(x + Q_1) \bar{Q}_0] \bar{Q}_0 + [(\bar{x} + Q_1) \bar{Q}_0] Q_0 = (x + Q_1) \bar{Q}_0$$

$$Q_1^* = x \bar{Q}_1 + \bar{x} Q_1 = x$$

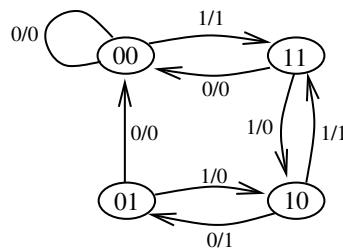
#### 3. Tabela de transição de estados

Cada célula da tabela representa o próximo estado e a saída  $((Y_1 Y_0)/z)$ .

Estado atual ( $Q_1 Q_0$ )	$Q_1^* Q_0^*/z$	
	$x = 0$	$x = 1$
00	00/0	11/1
01	00/0	10/0
10	01/1	11/1
11	00/0	10/0

#### 4. Diagrama de transição de estados

Cada nó representa um estado do sistema. Há uma aresta de um estado para outro se é possível uma transição de um para o outro. O rótulo nas arestas indica entrada  $x$  e saída  $z$  (leia-se  $x / z$ ). Como há apenas uma variável de entrada, que pode tomar os valores 0 ou 1, então há exatamente 2 arestas que saem de cada nó.



## 10.2 Projeto de circuitos sequenciais

Projeto de circuitos sequenciais é um processo inverso ao da análise. No entanto, o ponto de partida em geral não é uma tabela ou diagrama de estados, e sim uma descrição funcional do circuito. A partir da descrição funcional pode ser elaborado um diagrama transição de estados ou uma tabela de transição de estados. Algumas referências para este assunto: [Floyd, 2007] (seção 8.4 Projeto de contadores síncronos), [Hill and Peterson, 1993] (capítulo 10).

As etapas que fazem parte de projeto de circuitos sequenciais são:

- Descrição funcional
- Tabela de estados (que pode ser obtida a partir do diagrama de estados ou não)
- Tabela minimal de estados
- Tabela de transição
- Equação das entradas dos *flip-flops*
- Circuito

Esses conceitos serão abordados por meio de exemplos.

**Exemplo 1:** O primeiro exemplo explorado no estudo do processo de análise de circuitos foi o contador síncrono de 2 bits. Aqui vamos fazer o processo inverso, i.e., projetar o circuito partindo do diagrama de transição de estados.

Os estados possíveis no caso são 00, 01, 10 e 11. A transição desses estados deve ser cíclica, isto é,

00  $\rightarrow$  01  $\rightarrow$  10  $\rightarrow$  11  $\rightarrow$  00

Como temos duas variáveis de estado, vamos utilizar dois *flip-flops* JK. Precisamos entender então quais valores de  $J$  e  $K$  fazem as transições ocorrerem como gostaríamos.

Primeiro observemos quais são os possíveis valores de  $J$  e  $K$  que fazem o estado  $Q$  de um flip-flop passar para  $Q^*$ :

$Q \rightarrow Q^*$	$J$	$K$
$0 \rightarrow 0$	0	$\times$
$0 \rightarrow 1$	1	$\times$
$1 \rightarrow 0$	$\times$	1
$1 \rightarrow 1$	$\times$	0

A tabela acima mostra que, por exemplo, para ocorrer a transição  $0 \rightarrow 0$  (em  $Q \rightarrow Q^*$ ), o  $J$  deve ser 0 e o  $K$  pode ser tanto 0 quanto 1. Por quê? Se  $Q = 0$  e  $Q^* = 0$ , isso significa que ou o estado permaneceu como estava ( $J = K = 0$ ) ou foi “resetado” ( $J = 0$  e  $K = 1$ ).

A sequência de transições 00  $\rightarrow$  01  $\rightarrow$  10  $\rightarrow$  11  $\rightarrow$  00 pode ser descrita conforme a tabela a seguir:

$Q_1$	$Q_0$		$Q_1^*$	$Q_0^*$
0	0		0	1
0	1		1	0
1	0		1	1
1	1		0	0

Agora vamos deduzir a expressão do  $J$  e  $K$  de cada um dos *flip-flops*. Como temos dois *flip-flops*, denotemos por  $J_1$  e  $K_1$  as entradas e por  $Q_1$  o estado do *flip-flop* 1, e por  $J_0$  e  $K_0$  as entradas e por  $Q_0$  o estado do *flip-flop* 0. Note que as expressões que descrevem essas entradas dependem apenas das variáveis de estado  $Q_0$  e  $Q_1$ , já que não temos outras variáveis no circuito.

Como podemos definir a expressão para as entradas  $J_1$  e  $K_1$ ? Vamos primeiramente isolar  $Q_1^*$  na tabela de transição de estados. Teremos:

$Q_1$	$Q_0$		$Q_1^*$
0	0		0
0	1		1
1	0		1
1	1		0

Podemos ver que quando  $Q_1Q_0 = 00$ ,  $Q_1^* = 0$ . Neste caso, considerando apenas o *flip-flop* 1, a transição  $Q_1 \rightarrow Q_1^*$  que temos é  $0 \rightarrow 0$ . Tal transição pode ocorrer quando  $J_1 = 0$  e  $K_1 = X$ . Podemos aplicar o mesmo raciocínio para as demais linhas da tabela. Por exemplo, quando  $Q_1Q_0 = 01$ ,  $Q_1$  vai de 0 para 1 (e isto ocorre quando  $J_1 = 1$  e  $K_1 = X$ ); e assim por diante. Assim, para cada estado  $Q_1Q_0$  podemos definir  $J_1$  e  $K_1$  e portanto podemos pensar ambas como funções dependentes das variáveis  $Q_1Q_0$ , i.e.,  $J_1 = J_1(Q_1Q_0)$  e  $K_1 = K_1(Q_1Q_0)$ .

Para obter a expressão de  $J_1$  e de  $K_1$ , criamos um mapa de Karnaugh para cada um, no qual as variáveis são  $Q_1Q_0$  e os valores do mapa são os valores de  $J_1(Q_1Q_0)$  e  $K_1(Q_1Q_0)$ , respectivamente.

Mapa do $J_1$			Mapa do $K_1$					
\ $Q_0$		\ $Q_0$		\ $Q_0$				
$Q_1 \backslash$		0	1	$Q_1 \backslash$		0	1	
	+	-	-		+	-	-	
0		0		1		X		X
	+	-	-		+	-	-	
1		X		X		0		1

A expressão de  $J_1$  obtida a partir da minimização por mapa de Karnaugh é, portanto,  $J_1 = Q_0$ . Similarmente, temos também  $K_1 = Q_0$ .

Repetindo o mesmo processo para  $J_0$  e  $K_0$ , temos:

Mapa do $J_0$			Mapa do $K_0$					
\ $Q_0$		\ $Q_0$		\ $Q_0$				
$Q_1 \backslash$		0	1	$Q_1 \backslash$		0	1	
	+	-	-		+	-	-	
0		1		X		X		1
	+	-	-		+	-	-	
1		1		X		X		1

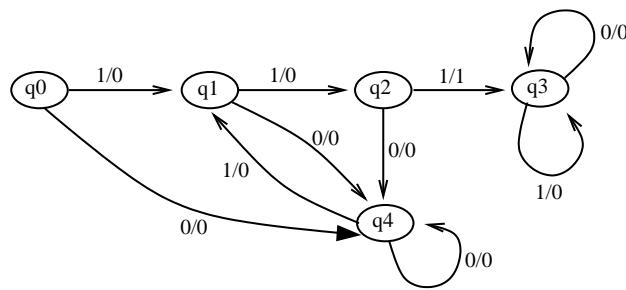
Após a minimização, temos  $J_0 = K_0 = 1$ .

Obtidas as expressões de  $J$  e  $K$  dos *flip-flops*, o circuito pode ser desenhado diretamente a partir das expressões. Compare as expressões obtidas aqui com as obtidas no processo de análise (veja também a figura 10.3). Ao se aplicar o processo de análise sobre o circuito, deveremos chegar ao diagrama de transição de estados que corresponde às transições  $00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00$  que foi o ponto de partida.

**Exemplo 2:** Este é um exemplo um pouco mais elaborado. Trata-se de um detector de início

de mensagem. Considere uma linha de transmissão de sinal, denotado por  $x$ , sincronizada com o *clock*. Uma ocorrência de 3 bits 1 consecutivos é considerado início de mensagem. Desejamos projetar um circuito síncrono que detecta um início de mensagem. Suponha que existe algum mecanismo que coloca o sistema detector de início de mensagem em um estado  $q_0$  a cada final de mensagem e suponha que inicialmente o sistema encontra-se no estado  $q_0$ .

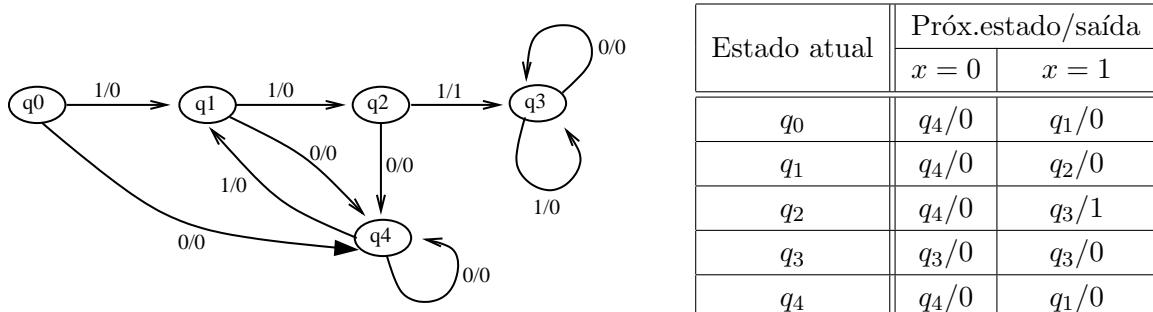
a) **Diagrama de estados:** uma possível forma de se começar o projeto de circuitos sequenciais é a construção de um diagrama de estados correspondente ao comportamento funcional desejado. Por exemplo:



Nesse diagrama, a detecção de início de mensagem corresponde a atingir o estado  $q_3$ .

Neste exemplo os estados são denotados genericamente por  $q_i$ , diferentemente do exemplo anterior. No caso anterior, por se tratar de um circuito bem conhecido, sabíamos de antemão quantos estados seriam suficientes.

b) **Tabela de estados:** o diagrama acima (reproduzido abaixo por conveniência) pode ser equivalentemente representado pela tabela de estados abaixo:



Note que a saída é 1 apenas quando o estado  $q_3$  é atingido.

c) **Tabela minimal de estados:** na tabela de estados acima, o estado  $q_0$  é equivalente ao estado  $q_4$ . Isso poderia ser percebido até no próprio diagrama de estados. No entanto, em geral nem sempre o diagrama de estados é gerado e, além disso, a equivalência de estados pode não ser tão

óbvia. De qualquer forma, nesta etapa reduz-se a tabela de estados a uma tabela minimal, ou seja, eliminam-se os estados equivalentes. Para não gerar confusão na identificação dos estados, na tabela minimal de estados é aconselhável a utilização de outros nomes para os estados. Desta forma, em vez da notação  $q_i$  para os estados, passaremos a utilizar as letras  $a, b, c, d$ . Fazendo  $a = q_4$ ,  $b = q_2$ ,  $c = q_1$  e  $d = q_3$  temos:

Estado	Prox.estado/saída	
	$x = 0$	$x = 1$
$a$	$a/0$	$c/0$
$b$	$a/0$	$d/1$
$c$	$a/0$	$b/0$
$d$	$d/0$	$d/0$

d) **Associação de estados:** se o número de estados na tabela minimal de estados é  $m$ , então serão necessários  $r$  flip-flops, onde  $r$  é tal que  $2^{r-1} < m \leq 2^r$ , para que o sistema seja capaz de representar esses  $m$  estados.

O problema de associação de estados consiste em definir qual das  $2^r$  combinações de valores binários será utilizado para representar cada um dos estados do sistema. Note que não consideramos isso no exemplo anterior, pois naquele caso os valores de estados estavam bem definidos. No exemplo que estamos considerando, como são 4 estados então são necessários  $r = 2$  flip-flops e existem as três seguintes possíveis associações:

Estados	Associação		
	1	2	3
$a$	00	00	00
$b$	01	11	10
$c$	11	01	01
$d$	10	10	11

As demais associações são equivalentes a um desses três no sentido de que correspondem a uma rotação vertical ou à complementação de uma ou ambas as variáveis e, portanto, em termos de circuito resultante teriam o mesmo custo.

e) **Tabelas de transição de estados:** para cada uma das associações consideradas, pode-se gerar uma tabela de transição de estados. Em cada uma das tabelas de transição abaixo, as atribuições de estado estão listadas seguindo a ordem do gray-code (00 – 01 – 11 – 10).

Associação 1				Associação 2			
Estado	$Q_1 Q_0$	$Q_1^* Q_0^*$		Estado	$Q_1 Q_0$	$Q_1^* Q_0^*$	
		$x = 0$	$x = 1$			$x = 0$	$x = 1$
$a$	00	00	11	$a$	00	00	01
$b$	01	00	10	$c$	01	00	11
$c$	11	00	01	$b$	11	00	10
$d$	10	10	10	$d$	10	10	10

Associação 3			
Estado	$Q_1 Q_0$	$Q_1^* Q_0^*$	
		$x = 0$	$x = 1$
$a$	00	00	01
$c$	01	00	10
$d$	11	11	11
$b$	10	00	11

f) **Equação das entradas dos *flip-flops*:** a equação das entradas dos *flip-flops* pode ser gerada a partir da análise das tabelas de transições, de forma análoga ao exemplo 1.

Iremos descrever esse processo novamente, considerando a associação 3 do item anterior. Primeiramente, observe que sabemos que do estado  $Q_1 Q_0$  o sistema irá para o estado  $Q_1^* Q_0^*$  e que cada variável de estado (no caso,  $Q_1$  e  $Q_0$ ) corresponde a um *flip-flop*. Assim, o que queremos descobrir é a expressão que descreve o sinal de entrada desses *flip-flops* para que a transição (mudança de estado) desejada ocorra.

Vamos analisar inicialmente o estado  $Q_1$ . Suponha que usaremos *flip-flops JK* neste circuito. Então, qual deve ser o valor de  $J_1$  e  $K_1$  para que a transição  $Q_1 \rightarrow Q_1^*$  ocorra? Para isso, recordemos a tabela do *flip-flop JK*. As tabelas abaixo descrevem, novamente, o comportamento do *flip-flop JK* (à esquerda) e as condições nas quais ocorre a transição  $Q \rightarrow Q^*$  (à direita). O símbolo  $\times$  indica uma entrada don't care.

$JK$	$Q^*$		$Q \rightarrow Q^*$	$J \quad K$	
	$Q = 0$	$Q = 1$		$0 \rightarrow 0$	$0 \times$
00	0	1	$0 \rightarrow 1$	1	$\times$
01	0	0	$1 \rightarrow 0$	$\times$	1
10	1	1	$1 \rightarrow 1$	$\times$	0
11	1	0			

Abaixo mostramos o mapa de  $J_1$  e  $K_1$  (em forma tabular), em função de  $Q_1 Q_0$  e  $x$ :

Associação 3 $Q_1 Q_0$	$Q_1$	$Q_1^*$		$J_1$		$K_1$	
		$x = 0$	$x = 1$	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a 00	0	0	0	0	0	$\times$	$\times$
c 01	0	0	1	0	1	$\times$	$\times$
d 11	1	1	1	$\times$	$\times$	0	0
b 10	1	0	1	$\times$	$\times$	1	0

As colunas  $Q_1$  e  $Q_1^*$  estão destacadas para evidenciar as transições (e, portanto, definir os possíveis valores de  $J_1$  e  $K_1$  que acarretam essas transições). Na primeira linha é indicada a transição de  $Q_1 = 0$  para  $Q_1^* = 0$  tanto quando  $x = 0$  como quando  $x = 1$ . Essa transição ocorre quando  $J_1 = 0$ , independente do valor de  $K_1$ . Na segunda linha, quando  $x = 0$ ,  $Q_1 = 0$  se mantém (i.e.,  $Q_1^* = 0$ ). Mas quando  $x = 1$ ,  $Q_1 = 0$  muda para  $Q_1^* = 1$  e isso ocorre quando  $J_1 = 1$ , independente do valor de  $K_1$ . E assim por diante.

Podemos agora fazer a minimização (mapa de Karnaugh) como feito no exemplo 1. No caso de  $J_1$ , o único 1 na coluna  $x = 1$  pode ser agrupado com o don't care  $\times$  logo abaixo dele e assim obtemos  $J_1 = x Q_0$ . No mapa de  $K_1$ , o também único 1 na coluna  $x = 0$  pode ser agrupado com o don't care  $\times$  logo abaixo dele e assim obtemos  $K_1 = \bar{x} \bar{Q}_0$ .

De forma análoga, repetimos o processo para  $Q_0$ . Restringindo a tabela de transição da associação 3 à variável  $Q_0$ , temos:

$Q_0$	$Q_0^*$	
	$x = 0$	$x = 1$
a 0	0	1
c 1	0	0
d 1	1	1
b 0	0	1

Portanto, as tabelas para  $J_0$  e  $K_0$  serão respectivamente

$Q_1 Q_0$	$J_0$		$Q_1 Q_0$	$K_0$	
	$x = 0$	$x = 1$		$x = 0$	$x = 1$
00	0	1	00	$\times$	$\times$
01	$\times$	$\times$	01	1	1
11	$\times$	$\times$	11	0	0
10	0	1	10	$\times$	$\times$

De onde obtemos que  $J_0 = x$  e  $K_0 = \bar{Q}_1$ .

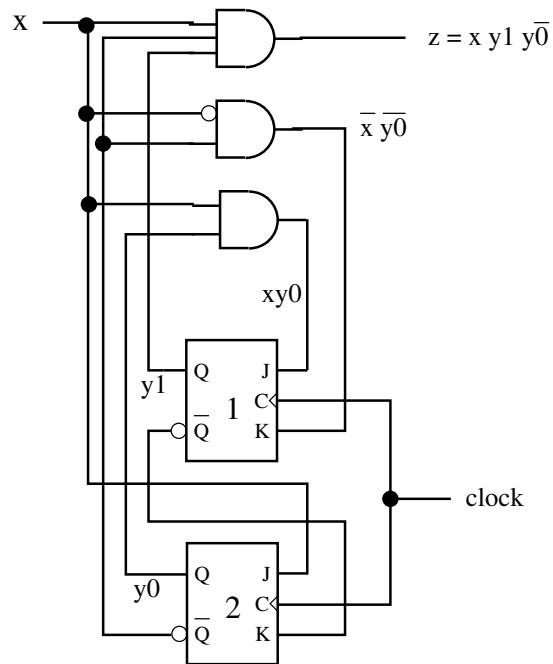
Além disso, a expressão para a saída  $z$  é dada por  $x y_1 \bar{Q}_0$  (pois existe uma única situação em

que a saída do circuito é 1; justamente quando ele se encontra no estado  $b$  e a entrada  $x$  é 1. A expressão segue do fato de termos associado ao estado  $b$  o par  $Q_1Q_0 = 10$ .

Procedimento similar pode ser aplicado para as associações 1 e 2. A associação 1 resulta em um circuito de custo (em termos de número total de portas lógicas) equivalente ao da associação 3 e a associação 2 resulta em um circuito de custo ligeiramente maior.

### g) O circuito!

As equações obtidas para a associação 3 correspondem ao seguinte circuito.



(refazer essa figura ...)

# Capítulo 11

## Organização de computadores

Última atualização em 06 de junho de 2018

Neste capítulo buscaremos entender como está organizado o *hardware* de um computador do ponto de vista lógico. Mais especificamente, estudaremos como um programa de computador é executado em um processador.

Um programa, antes de ser executado, precisa ser convertido para a linguagem de máquina do computador no qual ele será executado. A linguagem de máquina corresponde ao conjunto de instruções que podem ser executadas pelo computador, denominadas de instruções de máquina. Cada modelo de processador possui seu conjunto de instruções de máquina. Há dois conceitos relacionados a essa conversão. Um deles é **compilação** e refere-se ao processo de “traduzir” um programa escrito em uma linguagem de alto nível para um programa em linguagem de máquina. Um compilador recebe um programa-fonte (por exemplo, um programa em linguagem C) e gera um arquivo com o programa em linguagem de máquina, comumente chamados de programa binário ou ainda programa executável. Um programa binário é portanto uma sequência de instruções de máquina. A execução do programa consiste na execução sequencial dessas instruções. Já no caso de **interpretadores**, as instruções de máquina são geradas e enviadas ao processador uma a uma durante a execução do programa, não havendo portanto a geração de um arquivo com o código binário. Em geral os interpretadores atuam sobre uma linguagem intermediária (de baixo nível) não dependente de especificidades do *hardware*. Por exemplo, no caso da linguagem Python, ao se executar um programa, ocorre inicialmente um processo de conversão do programa-fonte para uma linguagem intermediária denominada *bytecode*. Em seguida, um interpretador Python converte os *bytecodes* em instruções de máquina, que são executadas pelo processador. A cada vez que esse mesmo programa é executado ocorre a conversão dos *bytecodes* para as respectivas instruções de máquina.

Quando um computador é ligado, alguns processamentos programados em *hardware* são executados. Entre esses processamentos está a ação de “carregar” e colocar em execução um sistema operacional. O sistema operacional (SO) é o programa que faz o gerenciamento do uso de recursos do computador, ou seja, garante que aplicativos (outros softwares) sejam executados em harmonia. Basicamente, cabe ao SO “carregar” para a memória do computador as instruções correspondentes aos diferentes programas a serem executados e colocar esses programas em uma fila de execução. Após um certo número de instruções de um programa ter sido executado, o SO trata de “interferir” para que a próxima instrução a ser executada seja do próximo programa na fila de execução. Desta forma, os vários programas na fila de execução são executados em esquema de revezamento, várias vezes, até que terminem. O ciclo de revezamento é tão breve que o usuário não percebe que o seu programa pode ter “parado” várias vezes.

Na figura a seguir é mostrada uma hierarquia que ilustra diferentes “camadas” entre um usuário e o *hardware*:

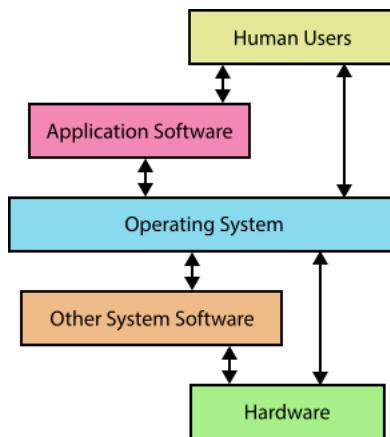


Imagen extraída de

[https://en.wikibooks.org/wiki/IB/Group\\_4/Computer\\_Science/Computer\\_Organisation](https://en.wikibooks.org/wiki/IB/Group_4/Computer_Science/Computer_Organisation)

Cada uma dessas camadas têm papel importante no funcionamento/uso de um computador. Neste capítulo, a camada de interesse é a de *hardware*. Portanto, não iremos estudar como um programa escrito em linguagem de alto nível é convertido para uma sequência de instruções de máquina, nem como sua execução é gerenciada pelo SO. Estudaremos os aspectos organizacionais do processador, relacionadas à execução de instruções de máquina, sob o ponto de vista lógico. Mais especificamente, examinaremos qual o papel dos circuitos lógicos na execução de instruções de máquina.

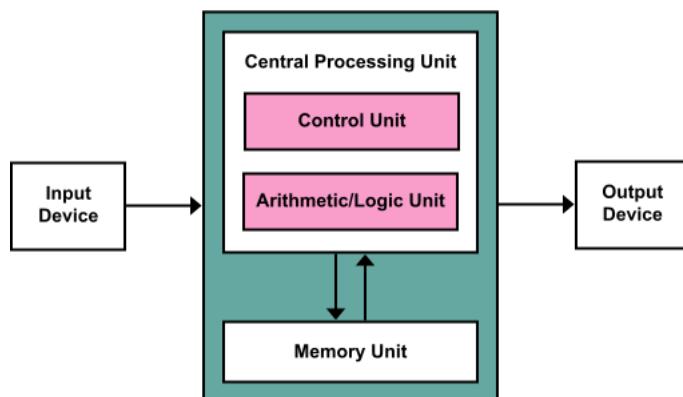
Na literatura da área, utilizam-se os termos “organização de computadores” e “arquitetura de computadores” para fazer referência aos aspectos relacionados à organização e funcionamento de computadores. De acordo com discussões relatadas em alguns fóruns, a distinção entre eles não é muito clara. Um dos entendimentos existentes é o seguinte:

- **Arquitetura de computadores:** refere-se a aspectos de desenho de um computador e especifica como é o funcionamento do *hardware* (conjunto de instruções, codificação das instruções, tipo de dados, forma de endereçamento, etc). Estabelece, portanto, um padrão que permite a comunicação entre as camadas superiores (por exemplo, sistemas operacionais) e o *hardware*. Os fabricantes de *hardware* implementam uma dada arquitetura; um sistema (*software*) que adere a essa arquitetura não depende dos detalhes de implementação.
- **Organização de computadores:** refere-se aos aspectos físicos do computador como o desenho dos circuitos, o tipo de memória, etc. São os aspectos que acabam influenciando custo, área do *chip*, eficiência, etc. A implementação de uma dada arquitetura pode considerar diferentes formas de organização de computadores.

Neste texto não nos preocuparemos em definir o que é arquitetura ou o que é organização, nem estudar arquiteturas reais. Estudaremos a organização de computadores usando um modelo de computação simples, com foco no estudo dos principais conceitos que fundamentam qualquer modelo de computação.

## 11.1 O modelo de computação de von Neumann

O precursor dos modelos de computadores atuais é o modelo proposto por *John von Neumann*. Um ponto notável no modelo de *von Neumann* é a revolucionária ideia de armazenar as instruções na memória do computador em vez de requerer alterações físicas no computador para cada tipo de instrução. Os componentes do modelo de *von Neumann* são mostrados na figura 11.1.



**Figura 11.1:** Componentes do modelo de von Neumann (Imagen extraída de [https://en.wikibooks.org/wiki/IB/Group\\_4/Computer\\_Science/Computer\\_Organisation](https://en.wikibooks.org/wiki/IB/Group_4/Computer_Science/Computer_Organisation)).

**Memória** É o módulo que armazena tanto instruções quanto dados. A memória, também conhecida por RAM (*Random Access Memory*) é uma coleção de palavras (múltiplos *bits* cada), identificadas por um endereço único. Podemos imaginar que essas palavras estão organizadas sequencialmente (linearmente), começando no endereço 0.

O tamanho da palavra (em termos de número de *bytes*) pode variar de computador para computador. São comuns palavras de 32 *bits* (4 *bytes*) ou de 64 *bits* (8 *bytes*). A magnitude de números inteiros que podem ser representados no computador é definida, portanto, pelo tamanho das palavras. Por exemplo, em um computador de 32 *bits*, pode-se representar  $2^{32}$  padrões binários distintos em uma palavra (portanto, se considerarmos números inteiros sem sinal, podemos representar números de 0 a  $2^{32} - 1$ ).

Os endereços são também representados com uma certa quantidade de *bits*; por exemplo, se considerarmos o endereçamento de 8 *bits*, os endereços que podem ser representados variam de 0 a  $2^8 - 1$ . Muitos computadores atuais permitem endereçamento *byte* a *byte*. Outra possibilidade é o endereçamento palavra a palavra. O endereçamento está diretamente relacionado ao acesso à memória do computador, tanto para leitura como para escrita. Tipicamente, uma operação de leitura de uma posição  $x$  da memória permite que se tenha acesso aos  $k$  *bytes* da memória a partir da posição  $x$ , onde  $k$  é o número de *bytes* de uma palavra.

Resumindo, a quantidade de *bits* das palavras define a magnitude dos números que podem ser representados no computador, enquanto o número de *bits* usados para endereçamento define a quantidade máxima de posições de memória que podem ser endereçadas.

Na RAM podem ser armazenados tanto dados como instruções.

**ULA** Unidade lógico-aritmética: como já vimos, é o módulo responsável pela execução (cálculo) das operações aritméticas e lógicas.

**Unidades de entrada e saída** São os módulos responsáveis por permitir que os dados “entrem” e “saiam” do computador. Sem esses módulos, não seria possível a um usuário “usar” um computador. Por exemplo, as ações executadas sobre um teclado ou com um *mouse* são apropriadamente tratados por esse módulo, gerando para cada ação uma representação interna adequada para o computador.

**Unidade de controle** Este é o módulo responsável por executar as instruções de um programa, em um processo conhecido por *Fetch-Execute Cycle* (isto é, buscar a próxima instrução a ser executada, decodificá-la e executá-la).

A unidade de controle e a ULA estão fortemente acopladas pois a execução de várias das instruções envolve a ULA. Assim, em geral usa-se o termo Unidade de Processamento Central para se referir ao conjunto “ULA + Unidade de controle”.

Os módulos acima estão conectados por linhas/canais de comunicação denominados barramentos

(*bus*, em inglês). Há basicamente três tipos de barramentos: *data bus* para transporte de dados, *address bus* para transporte de endereços e *control bus* para o transporte de informações de controle (por exemplo, para controlar se o *data bus* está transportando da direção da memória para o processador ou vice-versa). O *data bus* liga bidirecionalmente a CPU e a RAM, enquanto o *address bus* liga a CPU à memória.

Além disso, o processador utiliza registradores para armazenar informações importantes. Os tipos de registradores e seus papéis são descritos adiante.

## 11.2 Ciclo de instrução (*Fetch-Execute Cycle*)

Desde o momento em que é ligado até o momento em que é desligado, um computador faz apenas uma tarefa repetidamente. Ele executa ciclos de instrução (em inglês, *Instruction cycle* ou *Fetch-Execute Cycle*). A cada ciclo uma instrução de máquina é executada. Um ciclo desses, em seu modo mais simples, consiste de 3 passos, e é conhecida por FDX (*Fetch-Decode-Execute*):

1. *Fetch*: Buscar a próxima instrução a ser executada;
2. *Decode*: Decodificar a instrução a ser executada;
3. *Execute*: Executar a instrução.

O endereço da próxima instrução a ser executada está armazenada em uma memória especial do processador (chamada genericamente de registrador), denominada *program counter* (PC). A instrução a ser executada, depois de buscada na memória, é armazenada em outro registrador especial denominado *Instruction register* (IR). O endereço que está no PC é enviado para *address bus*, para se fazer acesso à palavra da memória onde está a instrução. O dado (que nesse caso deve ser uma instrução) que se encontra na palavra acessada é enviada para o *data bus*, sendo em seguida armazenado na IR. Antes da execução da instrução propriamente dita, o conteúdo do PC é incrementado de 1 (de modo a apontar para a próxima posição de memória que é aquela que contém a próxima instrução a ser executada).

## 11.3 Projeto e simulação de um processador

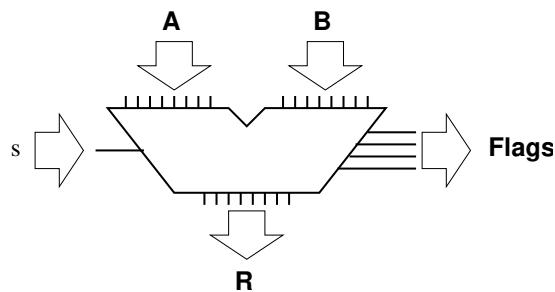
O estudo de organização de computadores envolverá o projeto e simulação de circuitos que compõem um processador. Parte dessas atividades já vem sendo realizadas ao longo do semestre na forma de EPs, utilizando o software Logisim (<http://www.cburch.com/logisim/>).

Até este momento estudamos um pouco da álgebra booleana, que permite escrevermos a relação entrada-saída de certos tipos de processamento de dados por meio de expressões booleanas. Essas, por sua vez, podem ser simplificadas por meio de manipulação simbólica baseada nas propriedades da álgebra booleana. Os circuitos lógicos são uma realização direta de uma expressão booleana; portanto, especificação e simplificação de expressões booleanas relacionam-se diretamente com o projeto da circuitaria de um processador.

Dentre os circuitos estudados, vale fazer a distinção entre os combinacionais e os sequenciais. No caso dos combinacionais, circuitos como somadores ou comparadores, ou ainda, MUX, DMUX, codificador e decodificador são componentes importantes de um processador. Os circuitos sequenciais relacionam-se com a capacidade do processador em reter um certo estado de processamento. Os circuitos devem ser projetados de forma a promover a transição correta de estados no sistema (por exemplo, instruções devem ser executadas na sequência correta; dados em uma determinada parte da memória devem ser acessados somente após terem sido efetivamente armazenados; etc).

### Características do processador a ser projetado e simulado:

- **ULA:** será utilizada para a realização das operações de adição e subtração, bem como de comparação. Sua especificação é conforme definida no EP2, i.e.,



As entradas  $A$  e  $B$  na parte superior ( $A = a_7 a_6 \dots a_1 a_0$  e  $B = b_7 b_6 \dots b_1 b_0$ ) correspondem aos dois números a serem operados, enquanto a saída  $R$  na parte inferior ( $R = r_7 r_6 \dots r_1 r_0$ ) corresponde ao resultado da operação (quando for o caso). Na lateral esquerda temos um pino seletor  $s$  que serve para indicar a operação aritmética a ser executada, e na lateral direita temos alguns *flags* de saída.

O seletor deve funcionar da seguinte forma:

$s$	Operação a ser executada
0	Adição ( $A + B$ )
1	Subtração ( $A - B$ )

Já as *flags* de saída consistem de 4 bits,  $o_3 o_2 o_1 o_0$ , sendo  $o_0$  a primeira e  $o_3$  a última de cima para baixo. As *flags* indicam os seguintes estados:

$$\begin{aligned}
 o_0 = 1 &\iff \text{overflow na operação aritmética} \\
 o_1 = 1 &\iff A > B \\
 o_2 = 1 &\iff A = B \\
 o_3 = 1 &\iff A < B
 \end{aligned}$$

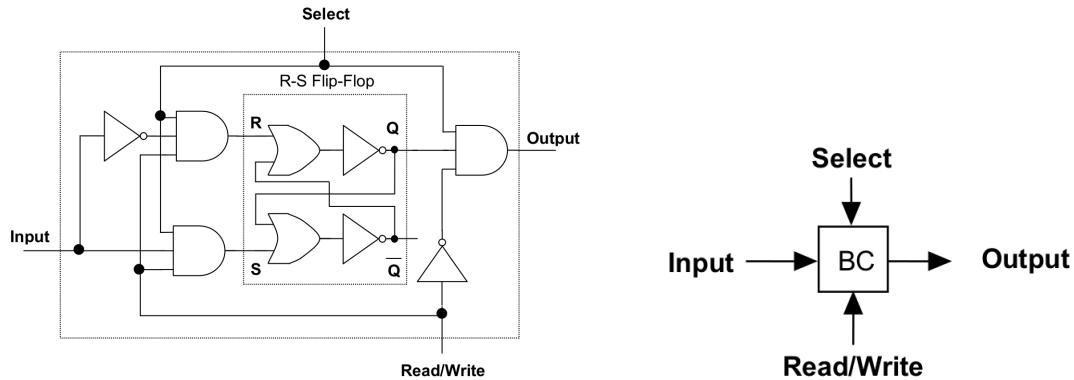
- Instrução: uma instrução é formada por 16 *bits*. Os 8 *bits* mais significativos correspondem ao código da instrução e os 8 *bits* menos significativos correspondem ao endereço referenciado pela instrução (quando pertinente).

As instruções que usaremos serão emprestadas do HIPO, um computador hipotético. Uma descrição do computador HIPO, incluindo o conjunto de instruções, pode ser encontrada em <https://www.ime.usp.br/~jstern/software/hipo/Hipo.pdf>.

O HIPO pode ser explorado por meio do Computador a papel (<https://www.ime.usp.br/~vwsetzer/comp-papel.html>) para a introdução de conceitos relacionados à computação, conforme feito em sala de aula.

- Contador: o PC (*Program counter*) pode ser um contador. Vamos supor que a primeira instrução estará sempre na posição de endereço zero na RAM. Os endereços serão todos de 8 *bits*.
- Registradores: o acumulador (ACC) e o registrador de instruções (IR) são ambos registradores utilizados pela UC. O ACC é tipicamente utilizado para armazenamento temporário de dados, geralmente em operações que envolvem a ULA. Os dados serão todos de 16 *bits*. O IR armazena temporariamente a instrução sendo executada no ciclo de instrução atual. O código da instrução é usado para acertar os sinais que controlam diferentes partes do processador para que a instrução seja executada corretamente. Caso a instrução envolva um endereço, esse deve ser enviado para o barramento de endereços.
- RAM: Cada posição da memória RAM consiste de palavras de 16 *bits*, podendo conter um dado ou uma instrução. Vamos considerar que os endereços são de 8 *bits* e, portanto, nossa memória terá 256 posições.

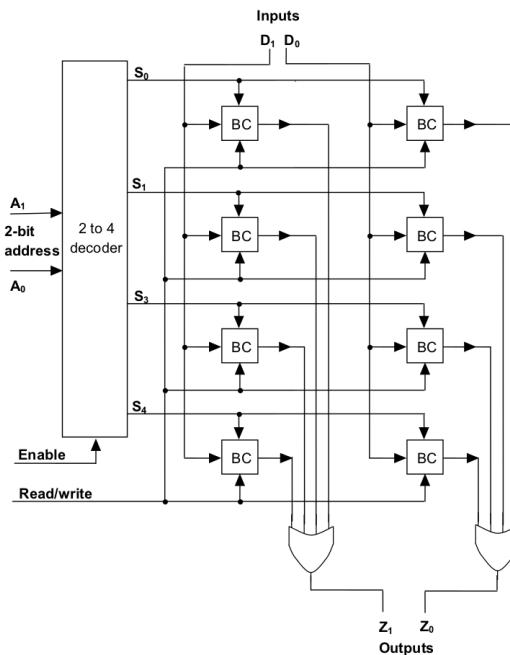
Uma posição é composta por 16 células binárias como a ilustrada a seguir. Cada célula binária do tipo ilustrado abaixo à esquerda pode armazenar exatamente 1 *bit*. À direita está o diagrama “caixa-preta” dela.



<http://watson.latech.edu/book/circuits/circuitsMicrocomputer3.html>

- **Read/Write = 0**  $\Rightarrow$  leitura      **Read/Write = 1**  $\Rightarrow$  escrita
- **select = 1** célula fica habilitada para a operação de **Read/Write**
- **input=1:** se **Read/Write=1** e **select=1** a entrada set do *flip-flop SR* (estado do SR vai para 1)
- **input=0:** se **Read/Write=1** e **select=1** a entrada reset do *flip-flop SR* (estado do SR vai para 0)

A memória RAM pode então ser vista como uma coleção dessas células. No nosso caso, cada posição de memória (palavra) consiste de 16 dessas células. No diagrama a seguir, ilustra-se o esquema de uma RAM com 4 posições (endereços de 2 bits e portanto endereços variando de 0 a 3) e palavras de apenas 2 bits. Note que o endereço que alimenta o decodificador ativa apenas uma das saídas, justamente para selecionar a palavra que corresponde à posição acessada.



Para executar uma operação de escrita numa posição, deve-se acertar o valor de *input*, colocar o valor de *Read/Write* em 1 e fazer o valor de *select* ser igual a 1 (este último depende do endereço na entrada do decodificador).

## 11.4 Detalhamento do ciclo de instrução

Examinaremos o ciclo de instrução utilizando algumas das instruções do HIPO como exemplo. Por meio desses exames ilustraremos o que acontece em cada um dos passos num ciclo FDX. Vamos supor que em nossa CPU a mudança de estados ocorre na subida do sinal *clock* e que uma instrução é representada em 16 *bits*: os 8 *bits* mais significativos correspondem ao código da instrução e os 8 menos significativos a um endereço referenciado por ela (se for o caso).

---

**Exemplo 1:** Seja a instrução 0BEE

Copiar [EE] para o AC

Aqui [EE] significa “conteúdo que está na posição EE da memória” e AC representa o acumulador (um registrador especial que a UC utiliza para armazenar dados envolvidos na execução de uma instrução).

Suponha que essa instrução será a próxima a ser executada e encontra-se no endereço 00 da RAM.

Isto significa que:

- o endereço 00 deve estar armazenado no PC
- a porta de endereços da RAM deve estar recebendo o conteúdo do PC
- o pino R/W da RAM deve estar setado em R
- a saída de dados da RAM deve estar alimentando o IR

*Fetch:* o *fetch* será efetuado no primeiro pulso de *clock*. Isto é, é esperado que a instrução 0BEE seja copiada da RAM para o registrador IR. Neste pulso do *clock* deve ocorrer também o incremento do valor do PC (o que acontece se o incremento for realizado apenas ao final do ciclo, após o passo *Execute*?)

*Decode:* o passo *decode* é executado tão logo a instrução é copiada para o IR (este passo não requer um pulso de *clock*).

Os 8 *bits* mais significativos (no caso 0B) serão usados por um decodificador de instruções para preparar as *flags* de controle para a execução propriamente dita. Além disso, caso a instrução envolva um endereço, este deve ser enviado para a porta de endereços da RAM (pois a instrução

em questão fará acesso a uma posição da RAM, seja para leitura ou para escrita). No exemplo, o acesso será para leitura. E o dado lido deverá ser direcionado para o AC. O decodificador deve garantir que todas as partes do processador estão devidamente preparados para a execução da instrução.

*Execute:* Supondo que o decodificador tenha preparado os *flags* e direcionamento de dados de forma correta, no segundo pulso do *clock* ocorrerá a execução propriamente dita da instrução. No exemplo considerado, o número que está no endereço EE da RAM será copiado para o AC.

A UC precisa garantir que após essa execução, o ciclo volte ao estado inicial. O PC deverá estar apontando para a posição da memória que contém a próxima instrução a ser executada.

---

**Exemplo 2:** Desvio incondicional 33EE

Desvie para EE

Lição de casa ☺

---

**Exemplo 3:** Adição 15EE

Some [EE] com [AC] e guarde o resultado em AC

Lição de casa ☺

---

## Apêndice A

# Relações de Ordem Parciais

Seja  $A$  um conjunto não vazio. Uma **relação binária**  $R$  sobre  $A$  é um subconjunto de  $A \times A$ , isto é,  $R \subseteq A \times A$ . Se  $(x, y) \in R$ , denotamos a relação de  $x$  por  $y$  como sendo  $xRy$  (lê-se  $x$ -erre- $y$ ).

**Relação de ordem parcial:** Uma relação binária  $\leq$  sobre  $A$  é uma **ordem parcial** se ela é

1. (reflexiva)  $x \leq x$ , para todo  $x \in A$
2. (anti-simétrica) Se  $x \leq y$  e  $y \leq x$ , então  $x = y$ , para todo  $x, y \in A$
3. (transitiva) Se  $x \leq y$  e  $y \leq z$  então  $x \leq z$ , para todo  $x, y, z \in A$

Se  $\leq$  é uma ordem parcial em  $A$ , então a relação  $\geq$  definida por, para quaisquer  $x, y \in A$ ,  $x \geq y$  se e somente se  $y \leq x$ , é também uma ordem parcial em  $A$ .

**Observação:** Apenas uma curiosidade: uma relação de equivalência é bem parecida com uma relação de ordem parcial. A diferença está na segunda propriedade: ordens parciais satisfazem anti-simetria, enquanto relações de equivalência satisfazem simetria (i.e., se  $x \sim y$  então  $y \sim x$ , para todo  $x, y \in A$ ).

### A.1 Conjuntos parcialmente ordenados (posets)

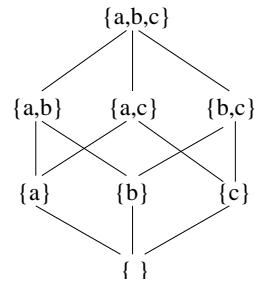
Um conjunto  $A$  munido de uma relação de ordem parcial  $\leq$  é denominado um conjunto parcialmente ordenado (ou *poset*) e denotado por  $(A, \leq)$ . Se  $(A, \leq)$  é um poset, então  $(A, \geq)$  também é um poset.

**Exemplo 1:** A relação de ordem  $\leq$  usual definida no conjunto dos números reais é uma ordem parcial (na verdade, ela é mais que uma ordem parcial; é uma **ordem total**, pois todos os elementos são comparáveis dois a dois). A relação  $<$  não é uma ordem parcial pois ela não é reflexiva.

**Exemplo 2:** A relação de inclusão de conjuntos  $\subseteq$  é uma ordem parcial.

**Diagrama de Hasse:** Escrevemos  $x < y$  quando  $x \leq y$  e  $x \neq y$ . Dado um poset  $(A, \leq)$  e  $x, y \in A$ , dizemos que  $y$  cobre  $x$  se, e somente se,  $x < y$  e não há outro elemento  $z \in A$  tal que  $x < z < y$ . Um diagrama de Hasse do poset  $(A, \leq)$  é uma representação gráfica onde vértices representam os elementos de  $A$  e dois elementos  $x$  e  $y$  são ligados por uma aresta se e somente se  $y$  cobre  $x$ . Em um diagrama de Hasse, os elementos menores (com relação a ordem parcial) são em geral desenhados abaixo dos elementos maiores.

**Exemplo:** O diagrama de Hasse do poset  $(\{a, b, c\}, \subseteq)$  é mostrado na figura A.1. Trata-se do cubo, visto no contexto de circuitos lógicos.



**Figura A.1:** Diagrama de Hasse de  $(\{a, b, c\}, \subseteq)$ .

# Referências Bibliográficas

- [Boole, 1854] Boole, G. (1854). *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities / by George Boole*. London :Walton and Maberly,. <http://www.biodiversitylibrary.org/bibliography/29413>. [25](#)
- [Brayton et al., 1984] Brayton, R. K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A. L. (1984). *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers. [69](#)
- [Coudert, 1994] Coudert, O. (1994). Two-level Logic Minimization: an Overview. *Integration, the VLSI Journal*, 17(2):97–140. [69](#)
- [Coudert, 1995] Coudert, O. (1995). Doing Two-level Minimization 100 Times Faster. In *Proc. of Symposium on Discrete Algorithms (SODA)*, San Francisco CA. [69](#)
- [Fišer and Hlavicka, 2003] Fišer, P. and Hlavicka, J. (2003). Boom - a heuristic boolean minimizer. *Computers and Informatics*, 22(1):19–51. [69](#)
- [Floyd, 2007] Floyd, T. L. (2007). *Sistemas Digitais - Fundamentos e Aplicações*. Bookman, nona edition. [88](#), [113](#)
- [Garnier and Taylor, 1992] Garnier, R. and Taylor, J. (1992). *Discrete Mathematics for New Technology*. Adam Hilger. [25](#)
- [Hill and Peterson, 1981] Hill, F. J. and Peterson, G. R. (1981). *Introduction to Switching Theory and Logical Design*. John Wiley, 3rd edition. [25](#)
- [Hill and Peterson, 1993] Hill, F. J. and Peterson, G. R. (1993). *Computer Aided Logical Design with Emphasis on VLSI*. John Wiley & Sons, fourth edition. [69](#), [113](#)
- [Hlavicka and Fiser, 2001] Hlavicka, J. and Fiser, P. (2001). BOOM - A Heuristic Boolean Minimizer. In *Proc. of ICCAD*, pages 439–442. [69](#)
- [McGreer et al., 1993] McGreer, P. C., Sanghavi, J., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1993). Espresso-Signature : A New Exact Minimizer for Logic Functions. *IEEE trans. on VLSI*, 1(4):432–440. [69](#)
- [Micheli, 1994] Micheli, G. D. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill. [69](#)
- [Shannon, 1938] Shannon, C. E. (1938). *A Symbolic Analysis of Relay and Switching Circuits*. [25](#)

[Whitesitt, 1961] Whitesitt, J. E. (1961). *Boolean Algebra and its Applications*. Addison-Wesley.

25